A3

An Interpretation of Standard ML in Type Theory

Robert Harper Christopher Stone

June 27Γ1997

CMU-CS-97-147

School of Computer Science
Carnegie Mellon University
PittsburghΓPA 15213

This report also appears as Fox Memorandum CMU-CS-FOX-97-01

19980106 048

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, in part by the National Science Foundation under Grant No. CCR-9502674, and in part by the US Army Research Office under Grant No. DAAH04-94-G-0289. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency, the U.S. Government or the National Science Foundation.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 4

Keywords: Standard MLΓtype theoryΓsemanticsΓprogramming language designΓcompilationΓtranslationΓelaborationΓstatic semanticsΓdynamic semantics

Abstract

We define an interpretation of Standard ML into type theory. The interpretation takes the form of a set of elaboration rules reminiscent of the static semantics given in *The Definition of Standard ML*. In particularΓthe elaboration rules are given in a relational styleΓexploiting indeterminacy to avoid over-commitment to specific implementation techniques. Elaboration consists of identifier scope resolutionΓtype checking and type inferenceΓexpansion of derived formsΓpattern compilationΓoverloading resolutionΓequality compilationΓand the coercive aspects of signature matching.

The underlying type theory is an explicitly-typed λ-calculus with productΓsumΓfunctionΓ and recursive typesΓtogether with module types derived from the translucent sum formalism of Harper and Lillibridge. Programs of the type theory are given a type-passing dynamic semantics compatible with constructs such as polymorphic equality that rely on type analysis at run-time.

This document supercedes the previous CMU CS technical reports CMU-CS-96-136 and CMU-CS-96-136R. The revision reflects our experience in implementing the specified elaboratorΓand includes several essential corrections and simplifications to the presentation.

Contents

1 Introduction

1.1 Overview

This document consists of a type-theoretic account of a variant of (Revised) Standard ML [MTHM97]Γhereafter referred to simply as Standard ML or SML. The approach taken here is to elaborate SML abstract syntax (the *external language*Γor EL) into an explicitly-typed λ-calculus (the *internal language*Γor IL). The internal language is designed to be as simple and orthogonal as possible while still being able to represent the entire Standard ML language.

The translation is presented by a set of inference rules reminiscent of the static semantics given in *The Definition of Standard ML*Γwith the internal language playing the role of the static semantic objects of *The Definition.* The translation rules typically define the translation of a phrase in terms of the translation of its constituent phrasesΓsubject to context-sensitive constraints expressed by the internal language type system. Context-sensitive formation constraints are expressed by type checking constraints on the translation. Type propagation is controlled by a combination of the translucent sum formalism together with the representation of abstract types as modules with opaque type components. The internal language ensures that abstraction is respectedΓandΓmoreoverΓprovides the requisite association of the abstract type to its representation at run-time.

We believe that looking at Standard ML language features in terms of how they translate into type theory clarifies a number of issues in the design of SML. For exampleΓthe notion of "type generativity" is replaced by explicit control of type propagation.

The internal language may be shared among many different language descriptions given in the style presented here. In particularΓwe envision the possibility of using the IL given here (with very minor modifications) for SchemeΓCaml Special LightΓor Haskell. A more long-term goal would be to see what changes would be needed—especially with regards to object-oriented features—in order to handle Java or Modula-3.

The elaboration has application to language implementation as wellΓas it may be viewed as a reference implementation of a front-end for an SML compiler. In particularΓwe are using this translation as a guide to our re-implementation of the TIL/ML compiler [HM95Γ Mor95ΓTar96ΓTMC$^+$96]. Compilers for other languages defined by interpretation into the internal language could share the back end of the TIL/ML compiler; only a front-end need be written for each specific language.

There are some disadvantages to our approach. The translation of some SML language features is quite complex. To some extent this is a direct reflection of the intrinsic complexity of mechanisms such as datatype definitions (which introduce several mutually recursive abstract typesΓeach a multinary sum of multinary product types) and polymorphicΓrecursive function bindings. HoweverΓensuring that the invariants of our encodings are respected everywhere in the document can be tedious and error-prone.

FurthermoreΓunderstanding the elaboration requires understanding the internal language as wellΓthough this may be mitigated if the same internal language can be used for several different external languages.

The type-theoretic interpretation of Standard ML is divided into three main parts.

Type Structure of the Internal Language. The internal language is an explicitly-typed λ-calculusΓwith a second-class modules system.

The core of the internal language is based on the XML and λ^{ML} calculi [HM93ΓHMM90]. The constructors of kind Ω (where Ω is the kind of types) include partial and total function typesΓrecord typesΓsum typesΓreference typesΓrecursive typesΓand a single extensible sum. AdditionallyΓthe constructors are extended with (restricted) tuples of constructors and functions at the constructor level. Note that there are no polymorphic types (polytypes) in our system.

The modules system is based on the translucent sum or manifest type modules calculi [HL94ΓLer94]. In addition to translucent signaturesΓwe have total and partial functor signatures. Our subtyping relation on signatures involves only forgetting of type definitions and totalityΓand not dropping or reordering components. This means that subtyping has no run-time effect.

The two levels are connected by the ability to define modules local to a core expression.

Dynamic Semantics of the Internal Language. The dynamic semantics is essentially a contextual semantics [WF91]. The handling of references and exceptions is similar to Harper's account of polymorphic references [Har93].

Elaboration. The translation is defined by a series of translation judgments of the general form

$$\Gamma \vdash \textit{EL-phrase} \leadsto \textit{IL-phrase} : \textit{IL-classifier}$$

where Γ is an internal-language context extended with information mapping EL identifiers to IL variables.

1.2 Notes on Implementation

This document is intended as an experiment in formally specifying the first stage of the TIL compilerΓand additionally to see if this generates a plausible definition for a language. We are not advocating that a compiler should generate exactly the IL code given hereΓnor even that the internal language of the compiler should be exactly the IL we present. HoweverΓany implementation of an elaborator should be essentially *equivalent* to what we presentΓand we suggest the compiler's internal language should be *definable* in terms of the IL we present here. Differences we would expect in an actual implementation include:

- The internal language is likely be extended to make our derived forms into primitivesΓpossibly along with other definable forms.
- A compiler is likely to use a much more complicated pattern compilerΓgenerating much more efficient pattern-matching code (in particularΓmaking better use case).
- Clausal function definitions would not be sequenced through uses of the exception mechanism.

- A compiler is likely to use a much more efficient implementation of elaboration contextsΓ avoiding the the sequential search which we have formalized.

- Equality functions would be cachedΓso that multiple equality tests for values of the same type would not generate multiple equality functions.

- In order to achieve SML's typing behaviorΓlocally-defined types at the module level are hidden through renaming rather than through use of internal-language local bindings. For simplicityΓall module-level local definitions (including values) are translated in this fashionΓthough this is not requiredΓand could cause wasted space at run-time. An implementation would use a more sophisticated criterion for which local bindings to rename.

1.3 Major Differences from Standard ML

1.3.1 Value Restriction

The value restriction used in this translation is different than that specified in SML.

- We treat

  $$\mathtt{val}\ \langle\mathtt{rec}\rangle\ \mathit{pat}_1\ \mathtt{=}\ \mathit{expr}_1\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathit{pat}_n\ \mathtt{=}\ \mathit{expr}_n$$

  as a derived form for

  $$\mathtt{val}\ \langle\mathtt{rec}\rangle\ \mathtt{(}\mathit{pat}_1\mathtt{,}\ldots\mathtt{,}\mathit{pat}_n\mathtt{)}\ \mathtt{=}\ \mathtt{(}\mathit{expr}_1\mathtt{,}\ldots\mathtt{,}\mathit{expr}_n\mathtt{)}.$$

- When `v` is a valueΓwe treat the binding `val h::t = v` as completely equivalent to `val h = hd v` and `val t = tl v`. Since in the latter cases `h` and `t` will not be generalized (`hd` and `tl` may raise exceptions!)Γ`h` and `t` are not made polymorphic in the former case.

  The translation allows variables in a `val` binding to be generalized if the translations of the bindings are "valuable." At the external language levelΓthis is equivalent to the right-hand side being a valueΓand the variable not be in the argument of a constructor patternΓunless the constructor in for a single-constructor datatype. Note that tuple-patterns pose no problemΓsince projection is a total function.

  This corresponds to the notion of "polymorphism as substitutionΓ" in which

  $$\mathtt{let\ val}\ \mathit{id}\mathtt{=}\mathit{expr}\ \mathtt{in}\ \mathit{expr}'\ \mathtt{end}$$

  is considered equivalent to $\{\mathit{expr}/\mathit{id}\}\mathit{expr}'$ when *expr* is a value.

  Because of this differenceΓa few variables that would have been polymorphic according to the SML specification will not be generalized in this treatment.

1.3.2 Abstype

We do not consider `abstype` declarations in this document because abstract data types can be easily created using the SML module system instead. There would be no technical difficulties in including `abstype`Γif desired.

1.3.3 Local and Higher-Order Functors

Our external language permits structure and functor declarations within a `let` or `local` declarationsΓand also within `structure` and `functor` declarations; this is a reflection of the fact that the internal language has local and higher-order modules.

These higher-order modules are based on the translucent sum formalism [HL94]. As this is compatible with first-class modules (which are not included in this definition)Γour higher-order functors propagate less type-sharing information than those in the system of MacQueen and Tofte [MT94]. Finding a clean type-theoretic description of this latter system is the subject of current research.

1.3.4 Top-level

Since we allow locally-defined modules in our external languageΓit suffices to consider programs to be closed expressions of type *ans*Γwhere *ans* is a fixed base type of answers. We do not explicitly consider the translation of SML top-level declarations.

Note that this does not require an *implementation* to accept only complete programs of some fixed type. For exampleΓa batch compiler may expect a sequence of structureΓfunctorΓ and signature bindings (SML top-level declarationsΓpossibly separately compiled)Γwhich are then "logically" treated as a sequence of declarations in a wrapper generated by the compiler. For a UNIX systemΓfor exampleΓthe wrapper might be along the lines of:

```
let ··· in 0 end handle _ => ~1
```

where *ans* is chosen to be the type `int` and a program returns either `0` for normal termination or `~1` for an uncaught exception.

To enable this interpretationΓwe view `signature` declarations as simple "macrosΓ" defining abbreviations for signatures. The translation assumes abbreviations have previously been expanded out in the program body.

1.4 Major Technical Differences from Version 2

1.4.1 Internal Language

Some minor changes have been made to the internal languageΓfor purposes of efficient implementation (e.g.Γthe unbundling of mutual recursion and projection operations at the constructor and expression levels)Γor for the purposes of a cleaner elaboration or internal language (e.g.Γthe generalization of the kind structureΓand the replacement of partial destructuring operations with total versions at the expression level).

1.4.2 Datatypes

The rule for datatypes has been simplified; additionally the internal language has been extended to allow all valid SML datatypes to elaborate. HoweverΓcertain datatypes that SML defines as admitting equality would require polymorphic recursion to implement this equality function. Since our language does not currently admit polymorphic recursion (which would correspond to recursive functors) these datatypes do not admit equality in our formulation.

1.4.3 Generativity

A technical problem with our presentation caused us to reject some legal Standard ML programsΓbecause of type information was not tracked properly. This problem involved abstract types defined in anonymous structure expressionsΓdefined locally in `let` or `local` at the module levelΓor hidden by SML's transparent ascription (`strexp : sigexp`). This has been corrected by:

1. Restricting the EL to named form at the module levelΓfollowing Leroy [Ler96]. This can always be achieved by a simple prepass over the program. (The grammar in Section 5 shows what we mean by named form.)

2. In the case that `let` and `local` module forms define abstract types locallyΓin the translation we augment the bodies of these forms with extra “hidden” fields containing these abstract types. In conjunction with the restriction to named formΓthis has the effect of causing types concealed by transparent ascription to be *renamed* rather than dropped entirely.[1]

Although the internal language module system presented here does not have most-specific signatures in generalΓwe also ensure that all modules generated by the elaborator do have most-specific signatures. This means that the elaborator never requires “guessing” of signatures for modules such as functor applications.

1.4.4 Subtyping

We have dropped the implicit subsumption between total and partial function types; every expression now has a unique most-specific types up to equivalence. Unfortunately this means datatype and exception constructors must now be handled specially when used as function values. HoweverΓa full subtyping relation would seem to be overkill—at least until features such as objects are added to the internal language.

1.4.5 Syntactic concatenation

Rather than using the starred structure convention for general concatenation of modulesΓwe use a syntactic concatenation of structure field bindings. This includes renaming to prevent ill-formed constructions where two fields share the same label.

[1] For simplicity, the rules essentially augment the bodies of `let` and `local` with *all* the locally-defined quantities. Most of these—including all value components and transparent type components—are obviously unnecessary and undesirable in an actual implementation.

2 Internal Language Abstract Syntax

2.1 Constructors and Kinds

$con ::=$	var	type variables
$\mid$	$\mathsf{Int} \mid \mathsf{Float} \mid \mathsf{Char} \mid \cdots$	base types
$\mid$	$\{rdecs\}$	record type
$\mid$	$con\ \mathsf{Ref}$	reference type
$\mid$	$con \rightharpoonup con'$	partial function type
$\mid$	$con \rightarrow con'$	total function type
$\mid$	Tagged	extensible sum type
$\mid$	$con\ \mathsf{Tag}$	exception-tag type
$\mid$	$\Sigma_{\langle lab \rangle}\ (lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)$	(labelled) sum type
$\mid$	$mod_{\mathsf{v}}.lab$	module projection
$\mid$	$\lambda var{:}knd.con$	constructor-level nonrecursive function
$\mid$	$\mu\ con$	constructor-level fixpoint
$\mid$	$con\ con'$	constructor application
$\mid$	$\{lab_1 = con_1, \ldots, lab_n = con_n\}$	records of constructors
$\mid$	$\pi_{lab}\ con$	record projection
$rdecs ::=$	$\cdot$	empty
$\mid$	$rdecs, rdec$	sequence
$rdec ::=$	$lab{:}con$	record field type
$knd ::=$	Ω	kind of types
$\mid$	$\{lab_1{:}knd_1, \ldots, lab_n{:}knd_n\}$	constructor tuple kinds
$\mid$	$knd \Rightarrow knd'$	constructor function kinds

Figure 1: Constructors and Kinds

The syntax $\langle \cdots \rangle$ denotes a phrase which may optionally appear.

2.2 Expressions

$exp ::=$	$scon$	constants
$\mid$	var	variables
$\mid$	loc	memory locations
$\mid$	tag	exception tags
$\mid$	$\mathsf{fix}\, fbnds\, \mathsf{end}$	mutually-recursive functions
$\mid$	$exp\ exp'$	application
$\mid$	$\{rbnds\}$	record expression
$\mid$	$\pi_{lab}\ exp$	record projection
$\mid$	$\mathsf{handle}\ exp\ \mathsf{with}\ exp'$	handle exception
$\mid$	$\mathsf{raise}^{con}\ exp$	raise exception
$\mid$	$\mathsf{ref}^{con}\ exp$	allocate new ref cell
$\mid$	$\mathsf{get}\ exp$	dereference
$\mid$	$\mathsf{set}\ (exp, exp')$	assignment
$\mid$	$\mathsf{roll}^{con}\ exp$	coerce into μ type
$\mid$	$\mathsf{unroll}\ exp$	coerce from μ type
$\mid$	$\partial\ exp$	coerce from partial to total function
$\mid$	$\mathsf{inj}^{con}_{lab}\ exp$	injection into sum
$\mid$	$\mathsf{proj}^{con}_{lab}\ exp$	total projection from sum
$\mid$	$\mathsf{case}^{con}\ exp\ \mathsf{of}\ exp_1, \ldots, exp_n\ \mathsf{end}$	sum case analysis
$\mid$	$\mathsf{new\_tag}[con]$	extend type Tagged
$\mid$	$\mathsf{tag}(exp, exp)$	injection into type Tagged
$\mid$	$\mathsf{iftagof}\ exp\ \mathsf{is}\ exp'\ \mathsf{then}\ exp''\ \mathsf{else}\ exp'''$	exception tag case analysis
$\mid$	$mod.lab$	module projection
$\mid$	$exp_1 =_{\mathsf{Int}} exp_2,\ exp_1 =_{\mathsf{Float}} exp_2, \ldots$	equalities at base types
$rbnds ::=$	$\cdot$	empty
$\mid$	$rbnds, rbnd$	sequence
$rbnd ::=$	$lab{=}exp$	record field binding
$fbnds ::=$	$\cdot$	empty
$\mid$	$fbnds, fbnd$	sequence
$fbnd ::=$	$var'(var{:}con){:}con' \mapsto exp$	function binding
$labs ::=$	$lab \mid labs.lab$	sequence of labels
$path ::=$	$var \mid var.labs$	path (qualified variable)

Figure 2: Expressions

2.3 Modules and Signatures

mod	::=	var	module variable
	|	$[sbnds]$	structure
	|	$\lambda var{:}sig.mod$	functor
	|	$mod\ mod'$	functor application
	|	$mod.lab$	projection from structure
	|	$mod{:}sig$	signature ascription
$sbnds$	::=	$\cdot$	structure field bindings
	|	$sbnds, sbnd$	
$sbnd$	::=	$lab \triangleright bnd$	
bnd	::=	$var{=}con$	constructor binding
	|	$var{=}exp$	expression binding
	|	$var{=}mod$	module binding
sig	::=	$[sdecs]$	structure signature
	|	$(var{:}sig) \rightharpoonup sig'$	partial functor signature
	|	$(var{:}sig) \rightarrow sig'$	total functor signature
$sdecs$	::=	$\cdot$	structure field declarations
	|	$sdecs, sdec$	
$sdec$	::=	$lab \triangleright dec$	
$decs$	::=	$\cdot$	declaration lists
	|	$decs, dec$	
dec	::=	$var{:}con$	expression variable declaration
	|	$var{:}sig$	module variable declaration
	|	$var{:}knd$	opaque type declaration
	|	$var{:}knd{=}con$	transparent type declaration
	|	$loc{:}con$	typed locations
	|	$tag{:}con$	typed exception tag

Figure 3: Modules and Signatures

2.4 Useful Abbreviations and Notation

- For readabilityΓwe often elide the internal names (*var*'s) when writing out *sbnds* (and *sdecs*). In all cases it should be immediately obvious how to consistently restore these with fresh variables.

- We use $sig \rightharpoonup sig'$ and $sig \rightarrow sig'$ to abbreviate $(var{:}sig) \rightharpoonup sig'$ and $(var{:}sig) \rightarrow sig'$ respectivelyΓwhere *var* is not free in sig'.

- Many grammatical classes (*labs*Γ*decs*Γ*rdecs*Γ*rbnds*Γ*fbnds*Γ*sdecs*Γ*sbnds*Γetc.) specify lists of elements. For each of these classes we define a binary append operationΓwritten with a comma. For exampleΓwe define

$$\begin{aligned} \mathit{decs}, \cdot \quad &:= \mathit{decs} \\ \mathit{decs}, (\mathit{decs}', \mathit{dec}') \quad &:= (\mathit{decs}, \mathit{decs}'), \mathit{dec}' \end{aligned}$$

with analogous definitions for all the other classes listed above.

- It is useful to define the following very general syntactic categories of program phrases and program phrase classifiers:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{phrase} & ::= & \mathit{exp} & \mathit{class} & ::= & \mathit{con} \\ & \mid & \mathit{mod} & & \mid & \mathit{sig} \\ & \mid & \mathit{con} & & \mid & \mathit{knd} \end{array}$$

- The notation $\{\mathit{phrase}/\mathit{var}\}\mathit{phrase}'$ denotes the capture-free substitution of *phrase* for free occurrences of *var* within *phrase'*.

- For consisionΓwe often abbreviate structure bindings $\mathit{lab}{\triangleright}\mathit{var}{=}\mathit{phrase}$ as $\mathit{lab}{=}\mathit{phrase}$Γ (and similarly signature bindings $\mathit{lab}{\triangleright}\mathit{var}{:}\mathit{class}$ as $\mathit{lab}{:}\mathit{class}$) when the local name is not used. In all cases it should be obvious how to consistently insert fresh variables to correct these omissions.

- We occasionally use the abbreviation $(\mathit{phrase}_i)_{i=1}^{n}$ as shorthand for $\mathit{phrase}_1, \cdots, \mathit{phrase}_n$ (where the phrases are comma-separated).

- The syntactic values of the language are defined in Figure 4. We view each class of syntactic values (class_v) as subsets of the corresponding class in the abstract syntax (class).

$$
\begin{array}{rcl}
\mathit{exp}_{\mathrm{v}} & ::= & \mathit{scon} \\
 & \mid & \mathit{loc} \\
 & \mid & \mathit{tag} \\
 & \mid & \mathit{path} \\
 & \mid & \{\mathit{rbnds}_{\mathrm{v}}\} \\
 & \mid & \mathsf{fix}\,\mathit{fbnds}\,\mathsf{end} \\
 & \mid & \pi_{\overline{k}}\,\mathsf{fix}\,\mathit{fbnds}\,\mathsf{end} \\
 & \mid & \mathsf{inj}_{\mathit{lab}}^{\mathit{con}}\ \mathit{exp}_{\mathrm{v}} \\
 & \mid & \mathsf{tag}(\mathit{exp}_{\mathrm{v}}, \mathit{exp}_{\mathrm{v}}') \\
 & \mid & \mathsf{roll}^{\mathit{con}}\ \mathit{exp}_{\mathrm{v}} \\
 & \mid & \partial\ \mathit{exp}_{\mathrm{v}} \\
\mathit{rbnds}_{\mathrm{v}} & ::= & \cdot \\
 & \mid & \mathit{rbnds}_{\mathrm{v}}, \mathit{rbnd}_{\mathrm{v}} \\
\mathit{rbnd}_{\mathrm{v}} & ::= & \mathit{lab}{=}\mathit{exp}_{\mathrm{v}} \\
 & & \\
\mathit{val} & ::= & \mathit{exp}_{\mathrm{v}} \\
 & \mid & \mathit{mod}_{\mathrm{v}} \\
 & \mid & \mathit{con}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{mod}_{\mathrm{v}} & ::= & \mathit{path} \\
 & \mid & [\mathit{sbnds}_{\mathrm{v}}] \\
 & \mid & \lambda \mathit{var}{:}\mathit{sig}.\mathit{mod} \\
 & & \\
\mathit{bnd}_{\mathrm{v}} & ::= & \mathit{var}{=}\mathit{exp}_{\mathrm{v}} \\
 & \mid & \mathit{var}{=}\mathit{mod}_{\mathrm{v}} \\
 & \mid & \mathit{var}{=}\mathit{con} \\
 & & \\
\mathit{sbnds}_{\mathrm{v}} & ::= & \cdot \\
 & \mid & \mathit{sbnds}_{\mathrm{v}}, \mathit{sbnd}_{\mathrm{v}} \\
 & & \\
\mathit{sbnd}_{\mathrm{v}} & ::= & \mathit{lab}{:}\mathit{bnd}_{\mathrm{v}}
\end{array}
$$

Figure 4: IL values

2.5 Bindings and Scope

We define the functions BV(·) and dom(·) for various bindingsΓdeclarationsΓand lists thereof:

Function	*Definition*
$\mathrm{BV}(dec)$	$\mathrm{BV}(var{:}con) = var$ $\mathrm{BV}(var{:}knd) = var$ $\mathrm{BV}(var{:}knd{=}con) = var$ $\mathrm{BV}(var{:}sig) = var$ $\mathrm{BV}(loc{:}con) = loc$ $\mathrm{BV}(tag{:}con) = tag$
$\mathrm{BV}(decs)$	$\mathrm{BV}(dec_1, \cdots, dec_n) = \{\mathrm{BV}(dec_1), \cdots, \mathrm{BV}(dec_n)\}$
$\mathrm{BV}(sdecs)$	$\mathrm{BV}(lab_1 \triangleright dec_1, \cdots, lab_n \triangleright dec_n) = \{\mathrm{BV}(dec_1), \cdots, \mathrm{BV}(dec_n)\}$
$\mathrm{BV}(bnd)$	$\mathrm{BV}(var{=}exp) = var$ $\mathrm{BV}(var{=}con) = var$ $\mathrm{BV}(var{=}mod) = var$
$\mathrm{BV}(sbnds)$	$\mathrm{BV}(lab_1 \triangleright bnd_1, \cdots, lab_n \triangleright bnd_n) = \{\mathrm{BV}(bnd_1), \cdots, \mathrm{BV}(bnd_n)\}$
$\mathrm{dom}(sdecs)$	$\mathrm{dom}(lab_1 \triangleright dec_1, \cdots, lab_n \triangleright dec_n) = \{lab_1, \cdots, lab_n\}$
$\mathrm{dom}(rdecs)$	$\mathrm{dom}(lab_1{:}con_1, \cdots, lab_n{:}con_n) = \{lab_1, \cdots, lab_n\}$

The scopes of bound variables are given by the following table:

Binding Phrase	*Bound Vars*	*Scope*
$\mathsf{fix}\, \mathit{fbnds}, \mathit{var}'(\mathit{var}{:}\mathit{con}){:}\mathit{exp}{\mapsto}, \mathit{fbnds}'\ \mathsf{end}$	var'	entire phrase
”	var	exp
$\lambda \mathit{var}{:}\mathit{knd}.\mathit{con}$	var	con
$\mathit{sbnd}, \mathit{sbnds}$	$\mathrm{BV}(\mathit{sbnd})$	sbnds
$\lambda \mathit{var}{:}\mathit{sig}.\mathit{mod}$	var	mod
$\mathit{sdec}, \mathit{sdecs}$	$\mathrm{BV}(\mathit{sdec})$	sdecs
$\mathit{var}{:}\mathit{sig} \rightharpoonup \mathit{sig}'$	var	sig'
$\mathit{var}{:}\mathit{sig} \rightarrow \mathit{sig}'$	var	sig'

We follow standard practice and identify all phrases which differ only with respect to bound variablesΓlocationsΓand exception names. We use the notation FV(*phrase*) to denote the set of free variables in *phrase*. A phrase is said to be *closed* if it has no free variables (though it may contain free locations or tags).

3 Static Semantics

3.1 Introduction

In this section we define the well-formedness and typing judgments for the internal language. Points of interest include:

- There are no metatsyntactic "semantic objects" in the sense of the *Definition.*
- The rules are explicitly formulated so that a judgment holds only if its constituents (declaration listsΓetc.) are well-formed.
- The *valuable* expressions always evaluate to a value without side-effectsΓreferencing the storeΓor raising exceptions. SimilarlyΓvaluable modules evaluate without side-effectsΓreferencing the storeΓor raising exceptions. The purpose of distinguishing total and partial functionsΓas well as total and partial functorsΓis to specify which function/functor applications are valuable.

3.2 Notation

- Within the static semanticsΓoptional elements are enclosed by single brackets $\langle \cdots \rangle$. Within a single ruleΓeither all such optional elements must occurΓor none.

3.3 Judgment Forms

Section	*Judgment...*	*Meaning...*
3.4.1	$\vdash \mathit{decs}\ \mathrm{ok}$	*decs* is well-formed
	$\mathit{decs} \vdash \mathit{dec}\ \mathrm{ok}$	*dec* is well-formed
3.4.2	$\mathit{decs} \vdash \mathit{bnd} : \mathit{dec}$	*bnd* has declaration *dec*
3.4.3	$\mathit{decs} \vdash \mathit{knd} : \mathsf{Kind}$	*knd* is well-formed
3.4.4	$\mathit{decs} \vdash \mathit{con} : \mathit{knd}$	*con* has kind *knd*
3.4.5	$\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}$	constructor equivalence at kind *knd*
3.4.6	$\mathit{decs} \vdash \mathit{exp} : \mathit{con}$	*exp* has type *con*
3.4.7	$\mathit{decs} \vdash \mathit{sdecs}\ \mathrm{ok}$	*sdecs* is well-formed
	$\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}$	*sig* is well-formed
3.4.8	$\mathit{decs} \vdash \mathit{sdecs} \leq \mathit{sdecs}'$	component-wise subtyping
	$\mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig}$	signature subtyping
3.4.9	$\mathit{decs} \vdash \mathit{sdecs} \equiv \mathit{sdecs}'$	component-wise equivalence
	$\mathit{decs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathsf{Sig}$	signature equivalence
3.4.10	$\mathit{decs} \vdash \mathit{sbnds} : \mathit{sdecs}$	*sbnds* has declaration list *sdecs*
	$\mathit{decs} \vdash \mathit{mod} : \mathit{sig}$	*mod* has signature *sig*
3.4.11	$\mathit{decs} \vdash \mathit{exp} \downarrow \mathit{con}$	*exp* is valuable with type *con*
	$\mathit{decs} \vdash \mathit{mod} \downarrow \mathit{sig}$	*mod* is valuable with signature *sig*
	$\mathit{decs} \vdash \mathit{exp} \downarrow$	
	$\mathit{decs} \vdash \mathit{sbnds} \downarrow$	
	$\mathit{decs} \vdash \mathit{mod} \downarrow$	

3.4 Inference Rules

3.4.1 Well-formed Declarations

$$\boxed{\vdash \mathit{decs}\ \mathbf{ok}}$$

$$\frac{}{\vdash \cdot\ \mathrm{ok}} \tag{1}$$

$$\frac{\mathit{decs} \vdash \mathit{dec}\ \mathrm{ok}}{\vdash \mathit{decs}, \mathit{dec}\ \mathrm{ok}} \tag{2}$$

$$\boxed{\mathit{decs} \vdash \mathit{dec}\ \mathbf{ok}}$$

$$\frac{\mathit{decs} \vdash \mathit{knd} : \mathsf{Kind} \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{knd}\ \mathrm{ok}} \tag{3}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathit{knd} \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{knd}{=}\mathit{con}\ \mathrm{ok}} \tag{4}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \Omega \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{con}\ \mathrm{ok}} \tag{5}$$

$$\frac{\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig} \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}\ \mathrm{ok}} \tag{6}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}'\,\mathsf{Ref} : \Omega \qquad \mathit{loc} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{loc}{:}\mathit{con}\ \mathrm{ok}} \tag{7}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}'\,\mathsf{Tag} : \Omega \qquad \mathit{tag} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{tag}{:}\mathit{con}\ \mathrm{ok}} \tag{8}$$

3.4.2 Well-formed Bindings

$$\boxed{\mathit{decs} \vdash \mathit{bnd} \ :\ \mathit{dec}}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathit{knd}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{con} \ :\ \mathit{var}{:}\mathit{knd}} \tag{9}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{con} \ :\ \mathit{var}{:}\mathit{knd}{=}\mathit{con}'} \tag{10}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{exp} \ :\ \mathit{var}{:}\mathit{con}} \tag{11}$$

$$\frac{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}' \qquad \mathit{decs} \vdash \mathit{sig}' \leq \mathit{sig} : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{mod} \ :\ \mathit{var}{:}\mathit{sig}} \tag{12}$$

3.4.3 Well-formed Kinds

$$\boxed{decs \vdash knd : \mathsf{Kind}}$$

$$\frac{}{decs \vdash \Omega : \mathsf{Kind}} \tag{13}$$

$$\frac{lab_1, \ldots, lab_n \text{ distinct} \qquad n \geq 0}{decs \vdash \{lab_1{:}knd_1, \ldots, lab_n{:}knd_n\} : \mathsf{Kind}} \tag{14}$$

$$\frac{decs \vdash knd : \mathsf{Kind} \qquad decs \vdash knd' : \mathsf{Kind}}{decs \vdash knd \Rightarrow knd' : \mathsf{Kind}} \tag{15}$$

3.4.4 Well-formed Constructors

$$\boxed{decs \vdash con : knd}$$

$$\frac{\begin{array}{c} \vdash decs \text{ ok} \\ decs = decs', var{:}knd\langle = con\rangle, decs'' \end{array}}{decs \vdash var : knd} \tag{16}$$

$$\frac{}{decs \vdash \mathsf{Tagged} : \Omega} \tag{17}$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con\ \mathsf{Ref} : \Omega} \tag{18}$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con\ \mathsf{Tag} : \Omega} \tag{19}$$

$$\frac{decs \vdash con : \Omega \qquad decs \vdash con' : \Omega}{decs \vdash con \rightharpoonup con' : \Omega} \tag{20}$$

$$\frac{decs \vdash con : \Omega \qquad decs \vdash con' : \Omega}{decs \vdash con \rightarrow con' : \Omega} \tag{21}$$

$$\frac{\begin{array}{c} lab_1, \ldots, lab_n \text{ distinct} \\ decs \vdash con_1 : \Omega \quad \cdots \quad decs \vdash con_n : \Omega \end{array}}{decs \vdash \{lab_1{:}con_1, \ldots, lab_n{:}con_n\} : \Omega} \tag{22}$$

$$\frac{\begin{array}{c} \langle i \in 1..n \rangle \\ lab_1, \ldots, lab_n \text{ distinct} \\ decs \vdash con_1 : \Omega \quad \cdots \quad decs \vdash con_n : \Omega \end{array}}{decs \vdash \Sigma_{\langle lab_i \rangle}\,(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n) : \Omega} \tag{23}$$

$$\frac{\begin{array}{c} lab_1, \ldots, lab_n \text{ distinct} \qquad n \geq 0 \\ decs \vdash con_1 : knd_1 \qquad \cdots \qquad decs \vdash con_n : knd_n \end{array}}{decs \vdash \{lab_1{=}con_1, \ldots, lab_1{=}con_n\} : \{lab_1{:}knd_1, \ldots, lab_1{:}knd_n\}} \tag{24}$$

$$\frac{\begin{array}{c} i \in 1..n \\ \mathit{decs} \vdash \mathit{con} : \{\mathit{lab}_1{:}\mathit{knd}_1, \ldots, \mathit{lab}_n{:}\mathit{knd}_n\} \end{array}}{\mathit{decs} \vdash \pi_{\mathit{lab}_i}\, \mathit{con} : \mathit{knd}_i} \tag{25}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{knd} \vdash \mathit{con} : \mathit{knd}'}{\mathit{decs} \vdash \lambda \mathit{var}{:}\mathit{knd}.\mathit{con} : \mathit{knd}{\Rightarrow}\mathit{knd}'} \tag{26}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathit{knd}{\Rightarrow}\mathit{knd}}{\mathit{decs} \vdash \mu\, \mathit{con} : \mathit{knd}} \tag{27}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathit{knd}'{\Rightarrow}\mathit{knd} \qquad \mathit{decs} \vdash \mathit{con}' : \mathit{knd}'}{\mathit{decs} \vdash \mathit{con}\ \mathit{con}' : \mathit{knd}} \tag{28}$$

$$\frac{\mathit{decs} \vdash \mathit{mod}_{\mathrm{v}} : [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}']}{\mathit{decs} \vdash \mathit{mod}_{\mathrm{v}}.\mathit{lab} : \mathit{knd}} \tag{29}$$

3.4.5 Constructor Equivalence

$$\boxed{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}}$$

$$\frac{\begin{array}{c} \vdash \mathit{decs}\ \mathrm{ok} \\ \mathit{decs} = \mathit{decs}', \mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{decs}'' \end{array}}{\mathit{decs} \vdash \mathit{var} \equiv \mathit{con} : \mathit{knd}} \tag{30}$$

Rule 30: The well-formedness judgment $\vdash \mathit{decs}$ ok guarantees that $\mathrm{FV}(\mathit{con}) \cap \mathrm{BV}(\mathit{decs}'') = \emptyset$.

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{mod}_{\mathrm{v}} : [\mathit{sdecs}, \mathit{lab}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs}'] \\ \mathrm{BV}(\mathit{sdecs}) \cap \mathrm{FV}(\mathit{con}) = \emptyset \end{array}}{\mathit{decs} \vdash \mathit{mod}_{\mathrm{v}}.\mathit{lab} \equiv \mathit{con} : \mathit{knd}} \tag{31}$$

Rule 31: The projection must yield a valid constructor with respect to the ambient context, *decs*. This can always be arranged through use of the self rules (Rules 101 and 102).

$$\frac{\mathit{decs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \Omega \qquad \mathit{decs} \vdash \mathit{con}_1' \equiv \mathit{con}_2' : \Omega}{\mathit{decs} \vdash \mathit{con}_1{\rightharpoonup}\mathit{con}_1' \equiv \mathit{con}_2{\rightharpoonup}\mathit{con}_2' : \Omega} \tag{32}$$

$$\frac{\mathit{decs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \Omega \qquad \mathit{decs} \vdash \mathit{con}_1' \equiv \mathit{con}_2' : \Omega}{\mathit{decs} \vdash \mathit{con}_1{\to}\mathit{con}_1' \equiv \mathit{con}_2{\to}\mathit{con}_2' : \Omega} \tag{33}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega}{\mathit{decs} \vdash \mathit{con}\,\mathsf{Ref} \equiv \mathit{con}'\,\mathsf{Ref} : \Omega} \tag{34}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega}{\mathit{decs} \vdash \mathit{con}\,\mathsf{Tag} \equiv \mathit{con}'\,\mathsf{Tag} : \Omega} \tag{35}$$

$$\frac{\begin{array}{c} \mathit{lab}_1, \ldots, \mathit{lab}_n\ \text{distinct} \\ \forall i \in 1..n : \quad \mathit{decs} \vdash \mathit{con}_i \equiv \mathit{con}_i' : \Omega \end{array}}{\mathit{decs} \vdash \{\mathit{lab}_1{:}\mathit{con}_1, \cdots, \mathit{lab}_n{:}\mathit{con}_n\} \equiv \{\mathit{lab}_1{:}\mathit{con}_1', \cdots, \mathit{lab}_n{:}\mathit{con}_n'\} : \Omega} \tag{36}$$

Rule 36: To be equivalent, two IL record types must have equivalent components with the same labels in the same order.

$$\frac{\begin{array}{c} i \in 1..n \\ \mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \{\mathit{lab}_1{:}\mathit{knd}_1, \ldots, \mathit{lab}_n{:}\mathit{knd}_n\} \end{array}}{\mathit{decs} \vdash \pi_{\mathit{lab}_i}\ \mathit{con} \equiv \pi_{\mathit{lab}_i}\ \mathit{con}' : \mathit{knd}_i} \tag{37}$$

$$\frac{\begin{array}{c} i \in 1..n \\ \mathit{decs} \vdash \mathit{con} \equiv \{\mathit{lab}_1{:}\mathit{con}_1, \ldots, \mathit{lab}_n{:}\mathit{con}_n\} : \{\mathit{lab}_1{:}\mathit{knd}_1, \ldots, \mathit{lab}_n{:}\mathit{knd}_n\} \end{array}}{\mathit{decs} \vdash \pi_{\mathit{lab}_i}\ \mathit{con} \equiv \mathit{con}_i : \mathit{knd}_i} \tag{38}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}{\Rightarrow}\mathit{knd}}{\mathit{decs} \vdash \mu\ \mathit{con} \equiv \mu\ \mathit{con}' : \mathit{knd}} \tag{39}$$

$$\frac{\begin{array}{c} \langle i \in 1..n \rangle \\ \mathit{decs} \vdash \mathit{con}_1 \equiv \mathit{con}'_1 : \Omega \quad \cdots \quad \mathit{decs} \vdash \mathit{con}_n \equiv \mathit{con}'_n : \Omega \end{array}}{\begin{array}{l} \mathit{decs} \vdash \Sigma_{\langle \mathit{lab}_i \rangle}\ (\mathit{lab}_1{\mapsto}\mathit{con}_1, \ldots, \mathit{lab}_n{\mapsto}\mathit{con}_n) \equiv \\ \qquad\quad \Sigma_{\langle \mathit{lab}_i \rangle}\ (\mathit{lab}_1{\mapsto}\mathit{con}'_1, \ldots, \mathit{lab}_n{\mapsto}\mathit{con}'_n) : \Omega \end{array}} \tag{40}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \mathit{knd}'{\Rightarrow}\mathit{knd} \\ \mathit{decs} \vdash \mathit{con}'_1 \equiv \mathit{con}'_2 : \mathit{knd}' \end{array}}{\mathit{decs} \vdash \mathit{con}_1\ \mathit{con}_2 \equiv \mathit{con}'_1\ \mathit{con}'_2 : \mathit{knd}} \tag{41}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{knd}' \vdash \mathit{con} : \mathit{knd} \qquad \mathit{decs} \vdash \mathit{con} : \mathit{knd}'}{\mathit{decs} \vdash (\lambda \mathit{var}{:}\mathit{knd}'.\mathit{con})\ \mathit{con}' \equiv \{\mathit{con}'/\mathit{var}\}\mathit{con} : \mathit{knd}} \tag{42}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathit{knd}}{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con} : \mathit{knd}} \tag{43}$$

$$\frac{\mathit{decs} \vdash \mathit{con}' \equiv \mathit{con} : \mathit{knd}}{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}} \tag{44}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd} \\ \mathit{decs} \vdash \mathit{con}' \equiv \mathit{con}'' : \mathit{knd} \end{array}}{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}'' : \mathit{knd}} \tag{45}$$

3.4.6 Well-formed Expressions

$$\boxed{\mathit{decs} \vdash \mathit{exp} : \mathit{con}}$$

$$\frac{\begin{array}{c} \vdash \mathit{decs}\ \mathrm{ok} \\ \mathit{decs} = \mathit{decs}', \mathit{var}{:}\mathit{con}, \mathit{decs}'' \end{array}}{\mathit{decs} \vdash \mathit{var} : \mathit{con}} \tag{46}$$

$$\frac{\begin{array}{c} \vdash \mathit{decs}\ \mathrm{ok} \\ \mathit{decs} = \mathit{decs}', \mathit{loc}{:}\mathit{con}, \mathit{decs}'' \end{array}}{\mathit{decs} \vdash \mathit{loc} : \mathit{con}} \tag{47}$$

$$\frac{\begin{array}{c}\vdash \mathit{decs}\ \mathrm{ok}\\ \mathit{decs} = \mathit{decs}', \mathit{tag}{:}\mathit{con}, \mathit{decs}''\end{array}}{\mathit{decs} \vdash \mathit{tag} : \mathit{con}} \tag{48}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}' \rightharpoonup \mathit{con} \qquad \mathit{decs} \vdash \mathit{exp}' : \mathit{con}'}{\mathit{decs} \vdash \mathit{exp}\ \mathit{exp}' : \mathit{con}} \tag{49}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}' \rightarrow \mathit{con} \qquad \mathit{decs} \vdash \mathit{exp}' : \mathit{con}'}{\mathit{decs} \vdash \mathit{exp}\ \mathit{exp}' : \mathit{con}} \tag{50}$$

$$\frac{\begin{array}{c}\forall i \in 1..n : \quad \mathit{decs}, (\mathit{var}'_j{:}\mathit{con}_j \rightharpoonup \mathit{con}'_j)_{j=1}^{n}, \mathit{var}_i{:}\mathit{con}_i \vdash \mathit{exp}_i : \mathit{con}'_i\\ \text{Rule 52 does not apply.}\end{array}}{\begin{array}{c}\mathit{decs} \vdash \mathsf{fix}\,(\mathit{var}'_i(\mathit{var}_i{:}\mathit{con}_i){:}\mathit{con}'_i \mapsto \mathit{exp}_i)_{i=1}^{n}\,\mathsf{end} :\\ \{1{:}\mathit{con}_1 \rightharpoonup \mathit{con}'_1, \ldots, n{:}\mathit{con}_n \rightharpoonup \mathit{con}'_n\}\end{array}} \tag{51}$$

$$\frac{\begin{array}{c}\mathit{var}' \notin \mathrm{FV}\,\mathit{exp}\\ \mathit{decs}, \mathit{var}{:}\mathit{con} \vdash \mathit{exp} \downarrow \mathit{con}'\end{array}}{\mathit{decs} \vdash \mathsf{fix}\,\mathit{var}'(\mathit{var}{:}\mathit{con}){:}\mathit{con}' \mapsto \mathit{exp}\,\mathsf{end} : \{1{:}\mathit{con} \rightarrow \mathit{con}'\}} \tag{52}$$

$$\frac{\begin{array}{c}\mathit{lab}_1, \cdots, \mathit{lab}_n\ \text{distinct}\\ \mathit{decs} \vdash \mathit{exp}_1 : \mathit{con}_1 \quad \cdots \quad \mathit{decs} \vdash \mathit{exp}_n : \mathit{con}_n\end{array}}{\mathit{decs} \vdash \{\mathit{lab}_1{=}\mathit{exp}_1, \cdots, \mathit{lab}_n{=}\mathit{exp}_n\} : \{\mathit{lab}_1{:}\mathit{con}_1, \cdots, \mathit{lab}_n{:}\mathit{con}_n\}} \tag{53}$$

Rule 53: For the record bindings to be well-typed with respect to the declarations, they must have the same labels in the same order (with no duplicate labels) and components must be well-typed elementwise.

$$\frac{\mathit{decs} \vdash \mathit{exp} : \{\mathit{rdecs}, \mathit{lab}{:}\mathit{con}, \mathit{rdecs}'\}}{\mathit{decs} \vdash \pi_{\mathit{lab}}\ \mathit{exp} : \mathit{con}} \tag{54}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con} \qquad \mathit{decs} \vdash \mathit{exp}' : \mathsf{Tagged} \rightharpoonup \mathit{con}}{\mathit{decs} \vdash \mathsf{handle}\ \mathit{exp}\ \mathsf{with}\ \mathit{exp}' : \mathit{con}} \tag{55}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathsf{Tagged} \qquad \mathit{decs} \vdash \mathit{con} : \Omega}{\mathit{decs} \vdash \mathsf{raise}^{\mathit{con}}\ \mathit{exp} : \mathit{con}} \tag{56}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \Omega}{\mathit{decs} \vdash \mathsf{new\_tag}[\mathit{con}] : \mathit{con}\ \mathsf{Tag}} \tag{57}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}}{\mathit{decs} \vdash \mathsf{ref}^{\mathit{con}}\ \mathit{exp} : \mathit{con}\ \mathsf{Ref}} \tag{58}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}\ \mathsf{Ref}}{\mathit{decs} \vdash \mathsf{get}\ \mathit{exp} : \mathit{con}} \tag{59}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{exp} : \mathit{con}\ \mathsf{Ref}\\ \mathit{decs} \vdash \mathit{exp}' : \mathit{con}\end{array}}{\mathit{decs} \vdash \mathsf{set}\,(\mathit{exp}, \mathit{exp}') : \mathsf{Unit}} \tag{60}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{con} \equiv (\pi_{\mathit{lab}}\,(\mu\,\mathit{con}'))\,\langle \mathit{con}''\rangle : \Omega \\ \mathit{decs} \vdash \mathit{exp} : (\pi_{\mathit{lab}}\,(\mathit{con}'\,(\mu\,\mathit{con}')))\,\langle \mathit{con}''\rangle \end{array}}{\mathit{decs} \vdash \mathsf{roll}^{\mathit{con}}\;\mathit{exp} : \mathit{con}} \tag{61}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{con} \equiv (\pi_{\mathit{lab}}\,(\mu\,\mathit{con}'))\,\langle \mathit{con}''\rangle : \Omega \\ \mathit{decs} \vdash \mathit{exp} : \mathit{con} \end{array}}{\mathit{decs} \vdash \mathsf{unroll}\;\mathit{exp} : (\pi_{\mathit{lab}}\,(\mathit{con}'\,(\mu\,\mathit{con}')))\,\langle \mathit{con}''\rangle} \tag{62}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con} \rightarrow \mathit{con}'}{\mathit{decs} \vdash \partial\,\mathit{exp} : \mathit{con} \rightharpoonup \mathit{con}'} \tag{63}$$

$$\frac{\begin{array}{c} i \in 1..n \\ \mathit{con} = \Sigma_{\mathit{lab}_i}\,(\mathit{lab}_1 \mapsto \mathit{con}_1, \ldots, \mathit{lab}_n \mapsto \mathit{con}_n) \\ \mathit{decs} \vdash \mathit{exp} : \mathit{con}_i \end{array}}{\mathit{decs} \vdash \mathsf{inj}^{\mathit{con}}_{\mathit{lab}_i}\;\mathit{exp} : \mathit{con}} \tag{64}$$

$$\frac{\begin{array}{c} i \in 1..n \\ \mathit{decs} \vdash \mathit{exp} : \Sigma_{\mathit{lab}_i}\,(\mathit{lab}_1 \mapsto \mathit{con}_1, \ldots, \mathit{lab}_n \mapsto \mathit{con}_n) \end{array}}{\mathit{decs} \vdash \mathsf{proj}^{\Sigma_{\mathit{lab}_i}(\mathit{lab}_1 \mapsto \mathit{con}_1, \ldots, \mathit{lab}_n \mapsto \mathit{con}_n)}_{\mathit{lab}_i}\;\mathit{exp} : \mathit{con}_i} \tag{65}$$

$$\frac{\begin{array}{c} n \geq 1 \\ \mathit{con} = \Sigma(\mathit{lab}_1 \mapsto \mathit{con}_1, \ldots, \mathit{lab}_n \mapsto \mathit{con}_n) \\ \mathit{decs} \vdash \mathit{exp} : \mathit{con} \\ \forall i \in 1..n : \quad \mathit{decs} \vdash \mathit{exp}_i : \Sigma_{\mathit{lab}_i}\,(\mathit{lab}_1 \mapsto \mathit{con}_1, \ldots, \mathit{lab}_n \mapsto \mathit{con}_n) \rightharpoonup \mathit{con}' \end{array}}{\mathit{decs} \vdash \mathsf{case}^{\mathit{con}}\;\mathit{exp}\;\mathsf{of}\;\mathit{exp}_1, \ldots, \mathit{exp}_n\;\mathsf{end} : \mathit{con}'} \tag{66}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}\;\mathsf{Tag} \qquad \mathit{decs} \vdash \mathit{exp}' : \mathit{con}}{\mathit{decs} \vdash \mathsf{tag}(\mathit{exp}, \mathit{exp}') : \mathsf{Tagged}} \tag{67}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{exp} : \mathsf{Tagged} \qquad \mathit{decs} \vdash \mathit{exp}' : \mathit{con}\;\mathsf{Tag} \\ \mathit{decs} \vdash \mathit{exp}'' : \mathit{con} \rightharpoonup \mathit{con}' \qquad \mathit{decs} \vdash \mathit{exp}''' : \mathit{con}' \end{array}}{\mathit{decs} \vdash \mathsf{iftagof}\;\mathit{exp}\;\mathsf{is}\;\mathit{exp}'\;\mathsf{then}\;\mathit{exp}''\;\mathsf{else}\;\mathit{exp}''' : \mathit{con}'} \tag{68}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{mod} : [\mathit{sdecs}, \mathit{lab}{:}\mathit{con}, \mathit{sdecs}'] \\ \mathrm{BV}(\mathit{sdecs}) \cap \mathrm{FV}(\mathit{con}) = \emptyset \end{array}}{\mathit{decs} \vdash \mathit{mod}.\mathit{lab} : \mathit{con}} \tag{69}$$

Rule 69: A projection is only well-formed if the result is typable. If *mod* is a value, the projection is always well-formed.

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}' \qquad \mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega}{\mathit{decs} \vdash \mathit{exp} : \mathit{con}} \tag{70}$$

3.4.7 Well-formed Signatures

$$\boxed{\mathit{decs} \vdash \mathit{sdecs}\ \mathbf{ok}}$$

$$\frac{\vdash \mathit{decs}\ \mathrm{ok}}{\mathit{decs} \vdash \cdot\ \mathrm{ok}} \tag{71}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{dec}\ \mathrm{ok} \\ \mathit{decs}, \mathit{dec} \vdash \mathit{sdecs}\ \mathrm{ok} \\ \mathit{lab} \notin \mathrm{dom}(\mathit{sdecs})\end{array}}{\mathit{decs} \vdash \mathit{lab} \triangleright \mathit{dec}, \mathit{sdecs}\ \mathrm{ok}} \tag{72}$$

$$\boxed{\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}}$$

$$\frac{\mathit{decs} \vdash \mathit{sdecs}\ \mathrm{ok}}{\mathit{decs} \vdash [\mathit{sdecs}] : \mathsf{Sig}} \tag{73}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sig}' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig} \rightharpoonup \mathit{sig}' : \mathsf{Sig}} \tag{74}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sig}' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig} \rightarrow \mathit{sig}' : \mathsf{Sig}} \tag{75}$$

3.4.8 Signature Subtyping

$$\boxed{\mathit{decs} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}'}$$

$$\frac{}{\mathit{decs} \vdash \cdot\ \leq\ \cdot} \tag{76}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{knd}{=}\mathit{con} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lab} \triangleright \mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs}\ \leq\ \mathit{lab} \triangleright \mathit{var}{:}\mathit{knd}, \mathit{sdecs}'} \tag{77}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig} \\ \mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}'\end{array}}{\mathit{decs} \vdash \mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}, \mathit{sdecs}\ \leq\ \mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}', \mathit{sdecs}'} \tag{78}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{lab}{:}\mathit{dec}\ \equiv\ \mathit{lab}{:}\mathit{dec}' \\ \mathit{decs}, \mathit{dec} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}'\end{array}}{\mathit{decs} \vdash \mathit{lab} \triangleright \mathit{dec}, \mathit{sdecs}\ \leq\ \mathit{lab} \triangleright \mathit{dec}', \mathit{sdecs}'} \tag{79}$$

$$\boxed{\mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig}}$$

$$\frac{\mathit{decs} \vdash \mathit{sdecs} \;\leq\; \mathit{sdecs}'}{\mathit{decs} \vdash [\mathit{sdecs}] \leq [\mathit{sdecs}'] : \mathsf{Sig}} \tag{80}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}'_1 \leq \mathit{sig}'_2 : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \rightharpoonup \mathit{sig}'_1 \leq \mathit{var}{:}\mathit{sig}_2 \rightharpoonup \mathit{sig}'_2 : \mathsf{Sig}} \tag{81}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}'_1 \leq \mathit{sig}'_2 : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \rightarrow \mathit{sig}'_1 \leq \mathit{var}{:}\mathit{sig}_2 \rightharpoonup \mathit{sig}'_2 : \mathsf{Sig}} \tag{82}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}'_1 \leq \mathit{sig}'_2 : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \rightarrow \mathit{sig}'_1 \leq \mathit{var}{:}\mathit{sig}_2 \rightarrow \mathit{sig}'_2 : \mathsf{Sig}} \tag{83}$$

3.4.9 Signature Equivalence

$$\boxed{\mathit{decs} \vdash \mathit{sdecs} \;\equiv\; \mathit{sdecs}'}$$

$$\frac{}{\mathit{decs} \vdash \cdot \;\equiv\; \cdot} \tag{84}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{knd} \vdash \mathit{sdecs} \;\equiv\; \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs} \;\equiv\; \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}'} \tag{85}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd} \quad \mathit{decs}, \mathit{var}{:}\mathit{knd}{=}\mathit{con} \vdash \mathit{sdecs} \;\equiv\; \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs} \;\equiv\; \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{con}', \mathit{sdecs}'} \tag{86}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega \quad \mathit{decs}, \mathit{var}{:}\mathit{con} \vdash \mathit{sdecs} \;\equiv\; \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}, \mathit{sdecs} \;\equiv\; \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}', \mathit{sdecs}'} \tag{87}$$

$$\frac{\mathit{decs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathit{knd} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sdecs} \;\equiv\; \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}, \mathit{sdecs} \;\equiv\; \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}'} \tag{88}$$

$$\boxed{\mathit{decs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathsf{Sig}}$$

$$\frac{\mathit{decs} \vdash \mathit{sdecs} \;\equiv\; \mathit{sdecs}'}{\mathit{decs} \vdash [\mathit{sdecs}] \equiv [\mathit{sdecs}'] : \mathsf{Sig}} \tag{89}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_1 \equiv \mathit{sig}_2 : \mathsf{Sig} \qquad \mathit{decs}, \mathit{var}{:}\mathit{sig}_1 \vdash \mathit{sig}'_1 \equiv \mathit{sig}'_2 : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \rightharpoonup \mathit{sig}'_1 \equiv \mathit{var}{:}\mathit{sig}_2 \rightharpoonup \mathit{sig}'_2 : \mathsf{Sig}} \tag{90}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_1 \equiv \mathit{sig}_2 : \mathsf{Sig} \qquad \mathit{decs}, \mathit{var}{:}\mathit{sig}_1 \vdash \mathit{sig}'_1 \equiv \mathit{sig}'_2 : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \rightarrow \mathit{sig}'_1 \equiv \mathit{var}{:}\mathit{sig}_2 \rightarrow \mathit{sig}'_2 : \mathsf{Sig}} \tag{91}$$

3.4.10 Well-formed Modules

$$\boxed{\mathit{decs} \vdash \mathit{sbnds} \;:\; \mathit{sdecs}}$$

$$\frac{}{\mathit{decs} \vdash \cdot : \cdot} \tag{92}$$

$$\frac{\mathit{decs} \vdash \mathit{bnd} \;:\; \mathit{dec} \qquad \mathit{decs}, \mathit{dec} \vdash \mathit{sbnds} \;:\; \mathit{sdecs}}{\mathit{decs} \vdash \mathit{lab} \triangleright \mathit{bnd}, \mathit{sbnds} \;:\; \mathit{lab} \triangleright \mathit{dec}, \mathit{sdecs}} \tag{93}$$

$$\boxed{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}}$$

$$\frac{\begin{array}{c} \vdash \mathit{decs}\ \mathrm{ok} \\ \mathit{decs} = \mathit{decs}', \mathit{var}{:}\mathit{sig}, \mathit{decs}'' \end{array}}{\mathit{decs} \vdash \mathit{var} : \mathit{sig}} \tag{94}$$

$$\frac{\mathit{decs} \vdash \mathit{sbnds} \;:\; \mathit{sdecs}}{\mathit{decs} \vdash [\mathit{sbnds}] : [\mathit{sdecs}]} \tag{95}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{mod} : \mathit{sig}'}{\mathit{decs} \vdash \lambda \mathit{var}{:}\mathit{sig}.\mathit{mod} : \mathit{var}{:}\mathit{sig} \rightharpoonup \mathit{sig}'} \tag{96}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{mod} \downarrow \mathit{sig}'}{\mathit{decs} \vdash \lambda \mathit{var}{:}\mathit{sig}.\mathit{mod} : \mathit{var}{:}\mathit{sig} \rightarrow \mathit{sig}'} \tag{97}$$

$$\frac{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}' \rightharpoonup \mathit{sig} \qquad \mathit{decs} \vdash \mathit{mod}' : \mathit{sig}'}{\mathit{decs} \vdash \mathit{mod}\,\mathit{mod}' : \mathit{sig}} \tag{98}$$

Rule 98: Only functors with non-dependent types may be applied. Dependencies can be eliminated by uses of the subtyping and equivalence rules. If the argument is a value, dependencies can always be eliminated.

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{mod} : [\mathit{sdecs}, \mathit{lab}{:}\mathit{sig}, \mathit{sdecs}'] \\ \mathrm{BV}(\mathit{sdecs}) \cap \mathrm{FV}(\mathit{sig}) = \emptyset \end{array}}{\mathit{decs} \vdash \mathit{mod}.\mathit{lab} : \mathit{sig}} \tag{99}$$

Rule 99: A projection is only well-formed if the result can be given a signature in the ambient context. If *mod* is a value, this can always occur.

$$\frac{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}}{\mathit{decs} \vdash \mathit{mod}{:}\mathit{sig} : \mathit{sig}} \tag{100}$$

Rule 100: Ascription of a signature to a module can make types in *mod* abstract by forgetting equations.

$$\frac{\mathit{decs} \vdash \mathit{mod}_{\mathrm{V}} : [\mathit{sdecs}, \mathit{lab} \triangleright \mathit{var}{:}\mathit{knd}, \mathit{sdecs}']}{\mathit{decs} \vdash \mathit{mod}_{\mathrm{V}} : [\mathit{sdecs}, \mathit{lab} \triangleright \mathit{var}{:}\mathit{knd}{=}\mathit{mod}_{\mathrm{V}}.\mathit{lab}, \mathit{sdecs}']} \tag{101}$$

Rule 101: The "self" rule. If *mod.labs* specifies a type and *mod* has a well-defined value then *mod.labs* ≡ *mod.labs*; we add this fact to the signature.

$$\frac{\begin{array}{c} decs \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright var{:}sig, sdecs'] \\ decs \vdash mod_{\mathrm{v}}.lab : sig' \end{array}}{decs \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright var{:}sig', sdecs']} \tag{102}$$

Rule 102: The "self" rule can be recursively applied.

$$\frac{decs \vdash mod : sig \qquad decs \vdash sig \leq sig' : \mathsf{Sig}}{decs \vdash mod : sig'} \tag{103}$$

3.4.11 Valuability

$$\boxed{decs \vdash exp \downarrow con}$$

$$\frac{decs \vdash exp : con \qquad decs \vdash exp \downarrow}{decs \vdash exp \downarrow con} \tag{104}$$

$$\boxed{decs \vdash mod \downarrow sig}$$

$$\frac{decs \vdash mod : sig \qquad decs \vdash mod \downarrow}{decs \vdash mod \downarrow sig} \tag{105}$$

$$\boxed{decs \vdash exp \downarrow}$$

$$\frac{}{decs \vdash exp_{\mathrm{v}} \downarrow} \tag{106}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \tag{107}$$

$$\frac{decs \vdash exp_1 \downarrow con' \rightarrow con \qquad decs \vdash exp_2 \downarrow}{decs \vdash exp_1\; exp_2 \downarrow} \tag{108}$$

$$\frac{decs \vdash exp_1 \downarrow \qquad \cdots \qquad decs \vdash exp_n \downarrow}{decs \vdash \{lab_1{=}exp_1, \cdots, lab_n{=}exp_n\} \downarrow} \tag{109}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \pi_{lab}\; exp \downarrow} \tag{110}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \mathsf{roll}^{con}\; exp \downarrow} \tag{111}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \mathsf{unroll}\; exp \downarrow} \tag{112}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \mathsf{inj}^{con}_{lab}\ exp \downarrow} \tag{113}$$

$$\frac{}{decs \vdash \mathsf{proj}^{con}_{lab}\ exp \downarrow} \tag{114}$$

$$\frac{decs \vdash exp \downarrow \qquad decs \vdash exp' \downarrow}{decs \vdash \mathsf{tag}(exp, exp') \downarrow} \tag{115}$$

$$\frac{\begin{array}{c} decs \vdash exp \downarrow \\ decs \vdash exp_1 \downarrow \quad \cdots \quad decs \vdash exp_n \downarrow \end{array}}{decs \vdash \mathsf{case}^{con}\ exp\ \mathsf{of}\ exp_1, \ldots, exp_n\ \mathsf{end} \downarrow} \tag{116}$$

$$\boxed{decs \vdash sbnds \downarrow}$$

$$\frac{}{decs \vdash \cdot \downarrow} \tag{117}$$

$$\frac{decs \vdash exp \downarrow con \qquad decs, var{:}con \vdash sbnds \downarrow}{decs \vdash lab \triangleright var{=}exp, sbnds \downarrow} \tag{118}$$

$$\frac{decs \vdash con : knd \qquad decs, var{:}knd{=}con \vdash sbnds \downarrow}{decs \vdash lab \triangleright var{=}con, sbnds \downarrow} \tag{119}$$

$$\frac{decs \vdash mod \downarrow sig \qquad decs, var{:}sig \vdash sbnds \downarrow}{decs \vdash lab \triangleright var{=}mod, sbnds \downarrow} \tag{120}$$

$$\boxed{decs \vdash mod \downarrow}$$

$$\frac{}{decs \vdash mod_{\mathsf{v}} \downarrow} \tag{121}$$

$$\frac{decs \vdash sbnds \downarrow}{decs \vdash [sbnds] \downarrow} \tag{122}$$

$$\frac{decs \vdash mod \downarrow sig' {\to} sig \qquad decs \vdash mod' \downarrow}{decs \vdash mod\ mod' \downarrow} \tag{123}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \tag{124}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod{:}sig \downarrow} \tag{125}$$

4 Dynamic Semantics

4.1 Introduction

The dynamic semantics of the internal language is a call-by-value operational semantics presented as a rewriting system on states of an abstract machine. The presentation is strongly influenced by the work of Plotkin [Plo81] and Wright and Felleisen [WF91]Γand is a significant departure from the framework employed in *The Definition*. In particular we employ a *small step* semantics in which transitions represent basic evaluation steps of an abstract machine. We rely on substitutionΓrather than environments; values are particular expressions of the language. Exception propagation is handled by explicitly maintaining the evaluation contextΓrather than relying on implicit rules for exception propagation. We maintain a store for assignable cellsΓas in *The Definition*Γand a typing context which types locations and dynamically-created tags associated with values of type Tagged; these are the only extra-linguistic constructs in the formalism. As Wright and Felleisen have demonstrated these could be made part of the language by introducing a `letref` construct.

Each state Σ of the abstract machine is a four-tuple of the form

$$(\Delta, \sigma, E, \mathit{phrase}),$$

where

- Δ is a typing context (*decs*) for locations and tags created at run-time. This maintains a record of what exception tags and locations have been allocatedΓand is also used in our soundness proofs.

- σ is a finite mapping from locations (*loc*'s) typed in Δ to expression values ($\mathit{exp}_{\mathrm{v}}$). The syntax for all internal language values appeared in Figure 4.

- E is a stack of evaluation frames (see Figure 5). This represents an implementation's control stackΓor equivalentlyΓthe current continuation. Appending two stacks corresponds to composition of continuations; accordinglyΓwe write $E \circ E'$ for the concatenation of E and E'Γdefined in the obvious way. The meta-variable R ranges over control stacks that do not contain a frame of the form "handle [] with *exp*".

- *phrase* is an expressionΓmoduleΓor constructor.

We let Σ_t range over the set of terminal statesΓwhere a state is called *terminal* if it has one of the forms:

$(\Delta, \sigma, [], \mathit{val})$	normal termination
$(\Delta, \sigma, [], \mathsf{raise}^{con}\ \mathit{exp}_{\mathrm{v}})$	uncaught exception

All other states are called *nonterminal*.

The dynamic semantics is a transition relation $\Sigma \hookrightarrow \Sigma'$ between states. As usualΓwe denote the reflexiveΓtransitive closure of $\hookrightarrow$ by $\hookrightarrow^*$.

$$
\begin{array}{rcl}
F & ::= & [\,]\ \mathit{exp} \\
 & \mid & \mathit{exp}_{\mathrm{v}}\ [\,] \\
 & \mid & \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}[\,], \mathit{rbnds}\} \\
 & \mid & \pi_{\mathit{lab}}\ [\,] \\
 & \mid & \textsf{handle}\ [\,]\ \textsf{with}\ \mathit{exp} \\
 & \mid & \textsf{raise}^{\mathit{con}}\ [\,] \\
 & \mid & \textsf{ref}^{\mathit{con}}\ [\,] \\
 & \mid & \textsf{get}\ [\,] \\
 & \mid & \textsf{set}\ ([\,], \mathit{exp}) \\
 & \mid & \textsf{set}\ (\mathit{exp}_{\mathrm{v}}, [\,]) \\
 & \mid & \textsf{roll}^{\mathit{con}}\ [\,] \\
 & \mid & \textsf{unroll}\ [\,] \\
 & \mid & \partial\ [\,] \\
 & \mid & \textsf{inj}_i^{\mathit{con}}\ [\,] \\
 & \mid & \textsf{proj}_i^{\mathit{con}}\ [\,] \\
 & \mid & \textsf{case}^{\mathit{con}}\ [\,]\ \textsf{of}\ \mathit{exp}_1, \ldots, \mathit{exp}_n\ \textsf{end} \\
 & \mid & \textsf{tag}(\mathit{exp}_{\mathrm{v}}, [\,]) \\
 & \mid & \textsf{iftagof}\ [\,]\ \textsf{is}\ \mathit{exp}'\ \textsf{then}\ \mathit{exp}''\ \textsf{else}\ \mathit{exp}''' \\
 & \mid & \textsf{iftagof}\ \mathit{exp}_{\mathrm{v}}\ \textsf{is}\ [\,]\ \textsf{then}\ \mathit{exp}''\ \textsf{else}\ \mathit{exp}''' \\
 & \mid & [\,].\mathit{lab} \\
 & \mid & [\mathit{sbnds}_{\mathrm{v}}, \mathit{lab}{\triangleright}\mathit{var}{=}[\,], \mathit{sbnds}] \\
 & \mid & [\,]\ \mathit{mod} \\
 & \mid & \mathit{mod}_{\mathrm{v}}\ [\,] \\
 & \mid & [\,]{:}\mathit{sig} \\
 \\
E & ::= & [\,] \\
 & \mid & E \circ F
\end{array}
$$

Figure 5: Evaluation Frames

4.2 Transition Relation

4.2.1 Search for Next Redex

$$(\Delta, \sigma, E, \mathit{exp}\ \mathit{exp}') \hookrightarrow (\Delta, \sigma, E \circ ([\,]\ \mathit{exp}'), \mathit{exp}) \tag{126}$$

$$(\Delta, \sigma, E \circ ([\,]\ \mathit{exp}), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E \circ (\mathit{exp}_{\mathrm{v}}\ [\,]), \mathit{exp}) \tag{127}$$

$$\begin{array}{l}(\Delta, \sigma, E, \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}\mathit{exp}, \mathit{rbnds}\}) \hookrightarrow \\ \quad (\Delta, \sigma, E \circ \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}[\,], \mathit{rbnds}\}, \mathit{exp}) \\ \qquad\qquad\qquad\qquad \text{if } \mathit{exp} \text{ is not a value}\end{array} \tag{128}$$

$$(\Delta, \sigma, E \circ \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}[\,], \mathit{rbnds}_{\mathrm{v}}'\}, \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}\mathit{exp}_{\mathrm{v}}, \mathit{rbnds}_{\mathrm{v}}'\}) \tag{129}$$

$$\begin{gathered}(\Delta, \sigma, E \circ \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}[], \mathit{rbnds}_{\mathrm{v}}{}', \mathit{lab}'{=}\mathit{exp}', \mathit{rbnds}\}, \mathit{exp}_{\mathrm{v}}) \hookrightarrow \\ (\Delta, \sigma, E \circ \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}\mathit{exp}_{\mathrm{v}}, \mathit{rbnds}_{\mathrm{v}}{}', \mathit{lab}'{=}[], \mathit{rbnds}\}, \mathit{exp}') \\ \text{if } \mathit{exp}' \text{ is not a value}\end{gathered} \tag{130}$$

$$\begin{gathered}(\Delta, \sigma, E, \pi_{\mathit{lab}}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\pi_{\mathit{lab}}\, []), \mathit{exp}) \\ \text{if } \mathit{exp}' \text{ is not a fix value}\end{gathered} \tag{131}$$

$$(\Delta, \sigma, E, \mathsf{handle}\, \mathit{exp}\, \mathsf{with}\, \mathit{exp}') \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{handle}\, []\, \mathsf{with}\, \mathit{exp}'), \mathit{exp}) \tag{132}$$

$$\begin{gathered}(\Delta, \sigma, E, \mathsf{raise}^{\mathit{con}}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{raise}^{\mathit{con}}\, []), \mathit{exp}) \\ \text{if } \mathit{exp} \text{ is not a value}\end{gathered} \tag{133}$$

$$(\Delta, \sigma, E \circ (\mathsf{raise}^{\mathit{con}}\, []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathsf{raise}^{\mathit{con}}\, \mathit{exp}_{\mathrm{v}}) \tag{134}$$

$$(\Delta, \sigma, E, \mathsf{ref}^{\mathit{con}}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{ref}^{\mathit{con}}\, []), \mathit{exp}) \tag{135}$$

$$(\Delta, \sigma, E, \mathsf{get}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{get}\, []), \mathit{exp}) \tag{136}$$

$$(\Delta, \sigma, E, \mathsf{set}\, (\mathit{exp}, \mathit{exp}')) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{set}\, ([], \mathit{exp}')), \mathit{exp}) \tag{137}$$

$$(\Delta, \sigma, E \circ (\mathsf{set}\, ([], \mathit{exp})), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{set}\, (\mathit{exp}_{\mathrm{v}}, [])), \mathit{exp}) \tag{138}$$

$$\begin{gathered}(\Delta, \sigma, E, \mathsf{roll}^{\mathit{con}}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{roll}^{\mathit{con}}\, []), \mathit{exp}) \\ \text{if } \mathit{exp} \text{ is not a value}\end{gathered} \tag{139}$$

$$(\Delta, \sigma, E \circ (\mathsf{roll}^{\mathit{con}}\, []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathsf{roll}^{\mathit{con}}\, \mathit{exp}_{\mathrm{v}}) \tag{140}$$

$$(\Delta, \sigma, E, \mathsf{unroll}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{unroll}\, []), \mathit{exp}) \tag{141}$$

$$\begin{gathered}(\Delta, \sigma, E, \partial\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\partial\, []), \mathit{exp}) \\ \text{if } \mathit{exp} \text{ is not a value}\end{gathered} \tag{142}$$

$$(\Delta, \sigma, E \circ (\partial\, []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \partial\, \mathit{exp}_{\mathrm{v}}) \tag{143}$$

$$\begin{gathered}(\Delta, \sigma, E, \mathsf{inj}^{\mathit{con}}_{\mathit{lab}}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{inj}^{\mathit{con}}_{\mathit{lab}}\, []), \mathit{exp}) \\ \text{if } \mathit{exp} \text{ is not a value}\end{gathered} \tag{144}$$

$$(\Delta, \sigma, E \circ (\mathsf{inj}^{\mathit{con}}_{\mathit{lab}}\, []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathsf{inj}^{\mathit{con}}_{\mathit{lab}}\, \mathit{exp}_{\mathrm{v}}) \tag{145}$$

$$(\Delta, \sigma, E, \mathsf{proj}^{\mathit{con}}_{\mathit{lab}}\, \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{proj}^{\mathit{con}}_{\mathit{lab}}\, []), \mathit{exp}) \tag{146}$$

$$\begin{aligned}(\Delta, \sigma, E, \mathsf{tag}(\mathit{exp}, \mathit{exp}')) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{tag}([], \mathit{exp}')), \mathit{exp})\\ \text{if } \mathit{exp} \text{ or } \mathit{exp}' \text{ is not a value}\end{aligned} \tag{147}$$

$$(\Delta, \sigma, E \circ \mathsf{tag}([], \mathit{exp}), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E \circ (\mathsf{tag}(\mathit{exp}_{\mathrm{v}}, [])), \mathit{exp}) \tag{148}$$

$$(\Delta, \sigma, E \circ (\mathsf{tag}(\mathit{exp}_{\mathrm{v}}, [])), \mathit{exp}_{\mathrm{v}}{}') \hookrightarrow (\Delta, \sigma, E, \mathsf{tag}(\mathit{exp}_{\mathrm{v}}, \mathit{exp}_{\mathrm{v}}{}')) \tag{149}$$

$$\begin{aligned}&(\Delta, \sigma, E, \mathsf{iftagof}\ \mathit{exp}\ \mathsf{is}\ \mathit{exp}'\ \mathsf{then}\ \mathit{exp}''\ \mathsf{else}\ \mathit{exp}''') \hookrightarrow\\ &\quad(\Delta, \sigma, E \circ (\mathsf{iftagof}\ []\ \mathsf{is}\ \mathit{exp}'\ \mathsf{then}\ \mathit{exp}''\ \mathsf{else}\ \mathit{exp}'''), \mathit{exp})\end{aligned} \tag{150}$$

$$\begin{aligned}&(\Delta, \sigma, E \circ (\mathsf{iftagof}\ []\ \mathsf{is}\ \mathit{exp}'\ \mathsf{then}\ \mathit{exp}''\ \mathsf{else}\ \mathit{exp}'''), \mathit{exp}_{\mathrm{v}}) \hookrightarrow\\ &\quad(\Delta, \sigma, E \circ (\mathsf{iftagof}\ \mathit{exp}_{\mathrm{v}}\ \mathsf{is}\ []\ \mathsf{then}\ \mathit{exp}''\ \mathsf{else}\ \mathit{exp}'''), \mathit{exp}')\end{aligned} \tag{151}$$

$$\begin{aligned}&(\Delta, \sigma, E, [\mathit{lab}{\triangleright}\mathit{var}{=}\mathit{phrase}, \mathit{sbnds}]) \hookrightarrow\\ &\quad(\Delta, \sigma, E \circ [\mathit{lab}{\triangleright}\mathit{var}{=}[], \mathit{sbnds}], \mathit{phrase})\end{aligned} \tag{152}$$

$$\begin{aligned}&(\Delta, \sigma, E \circ [\mathit{sbnds}_{\mathrm{v}}, \mathit{lab}{\triangleright}\mathit{var}{=}[],\\ &\qquad\qquad \mathit{lab}'{\triangleright}\mathit{var}'{=}\mathit{phrase}, \mathit{sbnds}], \mathit{val}) \hookrightarrow\\ &(\Delta, \sigma, E \circ [\mathit{sbnds}_{\mathrm{v}}, \mathit{lab}{\triangleright}\mathit{var}{=}\mathit{val},\\ &\qquad\qquad \mathit{lab}'{\triangleright}\mathit{var}'{=}[], \{\mathit{val}/\mathit{var}\}\mathit{sbnds}], \{\mathit{val}/\mathit{var}\}\mathit{phrase})\end{aligned} \tag{153}$$

$$(\Delta, \sigma, E \circ [\mathit{sbnds}_{\mathrm{v}}, \mathit{lab}{\triangleright}\mathit{var}{=}[]], \mathit{val}) \hookrightarrow (\Delta, \sigma, E, [\mathit{sbnds}_{\mathrm{v}}, \mathit{lab}{\triangleright}\mathit{var}{=}\mathit{val}]) \tag{154}$$

$$(\Delta, \sigma, E, \mathit{mod}.\mathit{lab}) \hookrightarrow (\Delta, \sigma, E \circ [].\mathit{lab}, \mathit{mod}) \tag{155}$$

$$(\Delta, \sigma, E, \mathit{mod}\ \mathit{mod}') \hookrightarrow (\Delta, \sigma, E \circ ([]\ \mathit{mod}'), \mathit{mod}) \tag{156}$$

$$(\Delta, \sigma, E \circ ([]\ \mathit{mod}'), \mathit{mod}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E \circ (\mathit{mod}_{\mathrm{v}}\ []), \mathit{mod}) \tag{157}$$

$$(\Delta, \sigma, E, \mathit{mod}{:}\mathit{sig}) \hookrightarrow (\Delta, \sigma, E \circ []{:}\mathit{sig}, \mathit{mod}) \tag{158}$$

4.2.2 Reduction

$$\begin{aligned}&(\Delta, \sigma, E \circ (\mathit{exp}_{\mathrm{v}}{}^{j}\ []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow\\ &\quad(\Delta, \sigma, E, \{\mathit{exp}_{\mathrm{v}}{}^{1}/\mathit{var}'_1\} \cdots \{\mathit{exp}_{\mathrm{v}}{}^{n}/\mathit{var}'_n\}\{\mathit{exp}_{\mathrm{v}}/\mathit{var}_j\}\mathit{exp}_j)\\ &\quad\ \text{where } \forall k \in 1..n :\\ &\quad\ \ \mathit{exp}_{\mathrm{v}}{}^{k} = \langle\partial\rangle\pi_{\overline{k}}\,\mathsf{fix}\,(\mathit{var}'_i(\mathit{var}_i{:}\mathit{con}_i){:}\mathit{con}'_i{\mapsto}\mathit{exp}_i)_{i=1}^{n}\ \mathsf{end}\end{aligned} \tag{159}$$

$$(\Delta, \sigma, E \circ (\pi_{\mathit{lab}}\ []), \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lab}{=}\mathit{exp}_{\mathrm{v}}, \mathit{rbnds}_{\mathrm{v}}{}'\}) \hookrightarrow (\Delta, \sigma, E, \mathit{exp}_{\mathrm{v}}) \tag{160}$$

$$(\Delta, \sigma, E \circ (\mathsf{handle}\ []\ \mathsf{with}\ \mathit{exp}), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathit{exp}_{\mathrm{v}}) \tag{161}$$

$$(\Delta, \sigma, E \circ (\mathsf{ref}^{con}\,[]), exp_{\mathrm{v}}) \hookrightarrow (\Delta[loc{:}con], \sigma[loc \mapsto exp_{\mathrm{v}}], E, loc) \quad \text{if } loc \notin \mathrm{BV}(\Delta) \tag{162}$$

$$(\Delta, \sigma, E \circ (\mathsf{get}\,[]), loc) \hookrightarrow (\Delta, \sigma, E, exp_{\mathrm{v}}) \quad \text{if } \sigma(loc) = exp_{\mathrm{v}} \tag{163}$$

$$(\Delta, \sigma, E \circ (\mathsf{set}\,(loc, [])), exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma[loc \mapsto exp_{\mathrm{v}}], E, \{\}) \tag{164}$$

$$(\Delta, \sigma, E \circ (\mathsf{unroll}\,[]), \mathsf{roll}^{con'}\, exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, exp_{\mathrm{v}}) \tag{165}$$

$$(\Delta, \sigma, E, \mathsf{new\_tag}[con]) \hookrightarrow (\Delta[tag{:}con\ \mathsf{Tag}], \sigma, E, tag) \quad \text{if } tag \notin \mathrm{BV}(\Delta) \tag{166}$$

$$(\Delta, \sigma, E \circ (\mathsf{proj}^{con'}_{lab'}\,[]), \mathsf{inj}^{con}_{lab}\, exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, exp_{\mathrm{v}}) \tag{167}$$

$$(\Delta, \sigma, E \circ (\mathsf{case}^{con}\,[]\ \mathsf{of}\ exp_1, \ldots, exp_n\ \mathsf{end}), \mathsf{inj}^{\Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)}_{lab_k}\, exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, exp_k\ exp_{\mathrm{v}}) \tag{168}$$

$$(\Delta, \sigma, E \circ (\mathsf{iftagof}\ \mathsf{tag}(tag, exp_{\mathrm{v}})\ \mathsf{is}\ []\ \mathsf{then}\ exp''\ \mathsf{else}\ exp'''), tag) \hookrightarrow (\Delta, \sigma, E, exp''\ exp_{\mathrm{v}}) \tag{169}$$

$$(\Delta, \sigma, E \circ (\mathsf{iftagof}\ \mathsf{tag}(tag, exp_{\mathrm{v}})\ \mathsf{is}\ []\ \mathsf{then}\ exp''\ \mathsf{else}\ exp'''), tag') \hookrightarrow (\Delta, \sigma, E, exp''') \quad \text{if } tag \neq tag' \tag{170}$$

$$(\Delta, \sigma, E \circ ((\lambda var{:}sig.mod)\,[]), mod_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \{mod_{\mathrm{v}}/var\}mod) \tag{171}$$

$$(\Delta, \sigma, E \circ []{:}sig, mod_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, mod_{\mathrm{v}}) \tag{172}$$

$$(\Delta, \sigma, E \circ (\mathsf{handle}\,[]\ \mathsf{with}\ exp') \circ R, \mathsf{raise}^{con}\, exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, exp'\ exp_{\mathrm{v}}) \tag{173}$$

$$(\Delta, \sigma, R, \mathsf{raise}^{con}\, exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, [], \mathsf{raise}^{ans}\, exp_{\mathrm{v}}) \tag{174}$$

5 External Language

5.1 Notation

As in the *Definition*Γoptional elements are enclosed by single brackets $\langle\cdots\rangle$ or double brackets $\langle\langle\cdots\rangle\rangle$. For the purposes of this grammarΓall optional choices are completely independent.

5.2 Grammar of the Abstract Syntax

The (disjoint) base syntax classes include *scon* (syntactic constants)Γ*tyvar* (type variables)Γ *id* (core-level identifiers)Γ*tycon* (type constructors)Γ*strid* (structure identifiers)Γand *funid* (functor identifiers). As in *The Definition of Standard ML*Γall but the first two have corresponding "long" formsΓcontaining a finite sequence of structure identifiers as as prefix.

$$\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{scon} \\
 & | & \mathit{longid} \\
 & | & \texttt{\{}\mathit{lab}_1 \texttt{ = } \mathit{expr}_1\texttt{,}\cdots\texttt{,}\mathit{lab}_n \texttt{ = } \mathit{expr}_n\texttt{\}} \\
 & | & \texttt{let } \mathit{strdec} \texttt{ in } \mathit{expr} \texttt{ end} \\
 & | & \mathit{expr}\ \mathit{expr}' \\
 & | & \mathit{expr} \texttt{ : } \mathit{ty} \\
 & | & \mathit{expr} \texttt{ handle } \mathit{match} \\
 & | & \texttt{raise } \mathit{expr} \\
 & | & \texttt{fn } \mathit{match} \\
 & | & \mathit{expr}_1 \texttt{ = } \mathit{expr}_2 \\
 & & \\
\mathit{mrule} & ::= & \mathit{pat} \texttt{ => } \mathit{expr} \\
\mathit{match} & ::= & \mathit{mrule} \\
 & | & \mathit{mrule} \texttt{ | } \mathit{match} \\
 & & \\
\mathit{strdec} & ::= & \cdot \\
 & | & \texttt{val (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{) } \mathit{pat} \texttt{ = } \mathit{exp} \\
 & | & \texttt{val (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{) rec } \mathit{pat} \texttt{ = } \mathit{exp} \\
 & | & \mathit{strdec}_1\ \mathit{strdec}_2 \\
 & | & \texttt{open } \mathit{longid}_1 \cdots \mathit{longid}_n \\
 & | & \texttt{exception } \mathit{id} \\
 & | & \texttt{exception } \mathit{id} \texttt{ of } \mathit{ty} \\
 & | & \texttt{exception } \mathit{id} \texttt{ = } \mathit{longid} \\
 & | & \texttt{local } \mathit{strdec}_1 \texttt{ in } \mathit{strdec}_2 \texttt{ end} \\
 & | & \texttt{type } \mathit{tybind} \\
 & | & \texttt{datatype } \mathit{datbind} \\
 & | & \texttt{datatype (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{) } \mathit{tycon} \texttt{ =} \\
 & & \quad \texttt{datatype (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{) } \mathit{longtycon} \\
 & | & \texttt{structure } \mathit{strbind} \\
 & | & \texttt{functor } \mathit{funbind}
\end{array}$$

$$\begin{array}{rcl}
\mathit{tybind} & ::= & (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{tybind} \rangle \\
\mathit{datbind} & ::= & (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind} \\
 & & \langle \texttt{and}\ \mathit{datbind} \rangle \\
\mathit{conbind} & ::= & \mathit{id}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle \langle \texttt{|}\ \mathit{conbind} \rangle \rangle
\end{array}$$

$$\begin{array}{rcl}
\mathit{strexp} & ::= & \mathit{longstrid} \\
 & | & \texttt{struct}\ \mathit{strdec}\ \texttt{end} \\
 & | & \mathit{longfunid}\ \texttt{(}\mathit{longstrid}\texttt{)} \\
 & | & \mathit{longstrid}\ \texttt{:}\ \mathit{sigexp} \\
 & | & \mathit{longstrid}\ \texttt{:>}\ \mathit{sigexp} \\
 & | & \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end}
\end{array}$$

$$\begin{array}{rcl}
\mathit{spec} & ::= & \cdot \\
 & | & \texttt{val}\ \mathit{id}\ \texttt{:}\ \mathit{ty} \\
 & | & \texttt{type}\ \mathit{typdesc} \\
 & | & \texttt{eqtype}\ \mathit{etypdesc} \\
 & | & \texttt{datatype}\ \mathit{datbind} \\
 & | & \texttt{datatype}\ (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{tycon}'\ \texttt{=} \\
 & & \quad \texttt{datatype}\ (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{longtycon} \\
 & | & \texttt{exception}\ \mathit{id} \\
 & | & \texttt{exception}\ \mathit{id}\ \texttt{of}\ \mathit{ty} \\
 & | & \texttt{structure}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp} \\
 & | & \texttt{functor}\ \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{:}\ \mathit{sigexp}' \\
 & | & \texttt{include}\ \mathit{sigexp} \\
 & | & \mathit{spec}_1\ \mathit{spec}_2 \\
 & | & \mathit{spec}\ \texttt{sharing type}\ \mathit{longid}_1\ \texttt{=}\ \mathit{longid}_2
\end{array}$$

$$\begin{array}{rcl}
\mathit{typdesc} & ::= & (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{tycon}\ \langle \texttt{and}\ \mathit{typdesc} \rangle \\
 & | & (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{typdesc} \rangle
\end{array}$$

$$\begin{array}{rcl}
\mathit{etypdesc} & ::= & (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{tycon}\ \langle \texttt{and}\ \mathit{etypdesc} \rangle
\end{array}$$

$$\begin{array}{rcl}
\mathit{sigexp} & ::= & \texttt{sig}\ \mathit{spec}\ \texttt{end} \\
 & | & \mathit{sigexp}\ \texttt{where type}\ (\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n)\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty}
\end{array}$$

$$\begin{array}{rcl}
\mathit{pat} & ::= & \mathit{scon} \\
 & | & \mathit{longid} \\
 & | & \_ \\
 & | & \mathit{pat}\ \texttt{:}\ \mathit{ty} \\
 & | & \mathit{longid}\ \mathit{pat} \\
 & | & \texttt{\{}\mathit{lab}_1\ \texttt{=}\ \mathit{pat}_1\texttt{,} \cdots \texttt{,}\mathit{lab}_n\ \texttt{=}\ \mathit{pat}_n\langle\texttt{,}\ \ldots\rangle\texttt{\}} \\
 & | & \mathit{pat}_1\ \texttt{as}\ \mathit{pat}_2 \\
 & | & \texttt{ref}\ \mathit{pat}
\end{array}$$

$$\begin{array}{rcl}
\mathit{ty} & ::= & \mathit{base} \\
 & \mid & \mathit{tyvar} \\
 & \mid & \texttt{\{}\mathit{lab}_1\ \texttt{:}\ \mathit{ty}_1\texttt{,}\cdots\texttt{,}\mathit{lab}_n\ \texttt{:}\ \mathit{ty}_n\texttt{\}} \\
 & \mid & \texttt{(}\mathit{ty}_1\texttt{,}\cdots\texttt{,}\mathit{ty}_n\texttt{)}\ \mathit{longtycon} \\
 & \mid & \mathit{ty}\ \texttt{->}\ \mathit{ty}' \\
 & & \\
\mathit{strbind} & ::= & \mathit{strid}\ \texttt{=}\ \mathit{strexp}\ \langle\textsf{and}\ \mathit{strbind}\rangle \\
\mathit{funbind} & ::= & \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{=}\ \mathit{strexp}\ \langle\textsf{and}\ \mathit{funbind}\rangle
\end{array}$$

5.3 Syntactic Restrictions

- No record expressionΓrecord patternΓor record type may contain duplicate field labels. No *tyvar* may appear more than once in a single sequence.

- Any type variable occuring in a *conbind* must also appear in the enclosing *datbind*. Any type variable appearing in the *ty* of a `where type` must appear in the type variable sequence.

- No `val`Γ`type`Γ`datatype`Γ`exception`Γ`structure`Γ`signature` or `functor` *strdec* or *spec* may bind the same identifier twice; this applies also to value constructors within a *datbind*.

- In a `val rec` declarationΓthe pattern must be of the form

  $$\texttt{\{}\mathit{lab}_1\texttt{=}\mathit{id}_1\texttt{,}\cdots\texttt{,}\mathit{lab}_n\texttt{=}\mathit{id}_n\texttt{\}}$$

  and the expression must be of the form

  $$\texttt{\{}\mathit{lab}_1\texttt{=fn}\ \mathit{match}_1\texttt{,}\cdots\texttt{,}\mathit{lab}_n\texttt{=fn}\ \mathit{match}_n\texttt{\}.}$$

  The "..." EL-notation may not appear in the pattern.

6 Elaboration

6.1 Introduction

In addition to type-checking and type-reconstructionΓthe elaborator performs the following tasks:

1. Datatypes are expanded into structures and signatures whose components include

   - an abstract type (implemented as a recursive sum type);
   - operations corresponding to datatype constructors as values (those datatype constructors which carry values are total functionsΓwhile non value-carrying constructors are constants of the abstract type);
   - an "expose" operation which presents datatype values as elements of a (tagged) sum type.

   The "generativity" of datatypes is handled via signature ascription; the type is made opaque and is therefore inequivalent to any previous type. The matching of datatypes in signatures reduces to the matching of substructures.

2. Polymorphism (including equality polymorphism) is encoded as a use of the modules system. Polymorphic values are translated into functorsΓwhich can be explicitly instantiated with structures of types (and equality functions). More preciselyΓthe functor takes a structure containing type constructors of kind Ω (for types which the polymorphic value requires to be equality typesΓthe structure also contains equality functions for the instantiating types); the functor returns a structure whose single component (with label "it") is the polymorphic value made monomorphic by instantiating it with the given types.

3. Equality polymorphismΓas just mentionedΓand equality types are handled by explicitly constructing and passing equality functions as needed. We do not explicitly distinguish types that admit equality; ratherΓa type admits equality iff the equality compiler can create an equality function for this type.

4. We make explicit the propagation of abstract types defined locally to module-level `let` and `local` constructorsΓor hidden through the use of transparent signature ascription. In particularΓthe "hiding" effect of these constructs on types is implemented as *renaming* rather than simple scoping. This is necessary to obtain the same type propagation behavior as the stamp-based semantics of *The Definition.*

5. Patterns are expanded into uses of the appropriate record projections and datatype deconstructors. Thus the elaboration specifies a reference pattern compiler.

6. Each series of external language bindings (*strdec*) elaborates into a structureΓcontaining a component for every variable bound in the external language. External language *identifiers* correspond to internal language *labels.*

7. Some structure labels are explicitly marked with an asterisk (*lab**). This indicates that the structure is "open" for the purposes of identifier lookup. (See the lookup rules for more details.)

8. All coercive aspects of the signature matching relation (the reordering and forgetting of components) are handled by introducing explicit coercion functors witnessing the relation. This makes the order and number of components of a structure apparent from its signatureΓthough there is a run-time cost to signature ascription.

6.2 Notation

- The overbar function $\overline{\cdot}$ maps each EL identifier to an IL label. We assume that this function is injectiveΓthat the range is coinfinite in the set of IL labelsΓand that identifiers of different classes map to different labels. In particularΓwe assume that the parser distinguishes between the classes of expression variablesΓtype constructorsΓ type variablesΓstructure identifiersΓsignature identifiersΓand functor identifiers. HoweverΓwe do not distinguish between an identifier being used as an expression variableΓ datatype constructorΓor exception constructor.

  The distinguished labels "eq"Γ"expose"Γ"it"Γand "tag" used by the elaborator are not in the range of the overbar mappingΓand other labels chosen to be fresh are similarly not in the range of the mapping.

  We extend the overbar mapping component-wise to long identifiersΓwhich thus map to sequences of labels.

- Optional elements are enclosed in single or double angle brackets. For each ruleΓeither all or none of the elements in single angle brackets must be presentΓand similarly all or none of the elements in double angle brackets must be present. Single and double angle brackets in the same rule represent two *independent* choices.

- In some casesΓthe optional element notation is insufficient. ThereforeΓwe have the additional notation

  $$\left\{ \begin{array}{c} \mathit{element}_1 \\ \text{or} \\ \mathit{element}_2 \end{array} \right\}$$

  which means that either *element*$_1$ or *element*$_2$ must be present. If there are multiple such choices in a single ruleΓthis means that either the first element should always be chosen in all casesΓor the second element must be chosen in all cases.

  An extension of this notation gives the choices subscripts. Then all choices with the same subscript must agree (all first element or all second element) but two choices with different subscripts are completely independent.

- The elaboration maintains an elaboration context ΓΓwhich is simply a list of structure declarations (*sdecs*) except that we allow duplicate labels. When Γ appears in an IL judgment where *decs* is expectedΓthere is an implicit coercion which drops all top-level labels and all signature declarations. We extend the notion of variable bindings from Section 2.5 with the following:

Function	*Definition*
$\mathrm{BV}(\Gamma)$	$\mathrm{BV}(\mathit{sdec},\Gamma) \;\; = \{\mathrm{BV}(\mathit{sdec})\} \cup \mathrm{BV}(\Gamma)$
$\mathrm{dom}(\Gamma)$	$\mathrm{dom}(\mathit{sdec},\Gamma) \;\; = \{\mathrm{dom}(\mathit{sdec})\} \cup \mathrm{dom}(\Gamma)$

Binding Phrase	*Bound Vars*	*Scope*
sdec,Γ	$\mathrm{BV}(\mathit{sdec})$	Γ

- We use an operation of "syntactic concatenation with renaming" for *sbnds* andΓin parallelΓfor *sdecs*. This is defined by:

$$\begin{array}{l}
(\cdot{+}{+}\mathit{sbnds}') : (\cdot{+}{+}\mathit{sdecs}') := \mathit{sbnds}' : \mathit{sdecs}' \\
(\mathit{lab}^{\langle * \rangle}\triangleright \mathit{bnd}, \mathit{sbnds}{+}{+}\mathit{sbnds}') : (\mathit{lab}^{\langle * \rangle}\triangleright \mathit{bnd}, \mathit{sdecs}{+}{+}\mathit{sdecs}') := \\
\quad \begin{cases} \mathit{lab}^{\langle * \rangle}\triangleright \mathit{bnd}, \mathit{sbnds}'' : \mathit{lab}^{\langle * \rangle}\triangleright \mathit{dec}, \mathit{sdecs}'' & \text{if } \mathit{lab}^{\langle * \rangle} \notin \mathrm{dom}(\mathit{sbnds}'') \\ \mathit{lab}'^{\langle * \rangle}\triangleright \mathit{bnd}, \mathit{sbnds}'' : \mathit{lab}'^{\langle * \rangle}\triangleright \mathit{dec}, \mathit{sdecs}'' & \text{otherwise}\Gamma\text{where } \mathit{lab}'^{\langle * \rangle} \notin \mathrm{dom}(\mathit{sbnds}'') \end{cases} \\
\quad \text{where } \mathit{sbnds}{+}{+}\mathit{sbnds}' : \mathit{sdecs}{+}{+}\mathit{sdecs}' = \mathit{sbnds}'' : \mathit{sdecs}''
\end{array}$$

6.3 Initial Basis

The elaborator assumes the presence of a structure $\mathit{basis}{:}\mathit{sig}_{\mathit{basis}}$ serving as the initial basis for the internal language. It must contain at least the following fields which define three exceptions:

$$\begin{array}{ll}
[\overline{\texttt{Bind}}^{*} & :[\mathsf{tag}{:}\mathsf{Unit\ Tag}, \overline{\texttt{Bind}}{:}\mathsf{Tagged}], \\
\overline{\texttt{Match}}^{*} & :[\mathsf{tag}{:}\mathsf{Unit\ Tag}, \overline{\texttt{Match}}{:}\mathsf{Tagged}], \\
\mathsf{fail}^{*} & :[\mathsf{tag}{:}\mathsf{Unit\ Tag}, \mathsf{fail}{:}\mathsf{Tagged}]].
\end{array}$$

6.4 Derived Forms

The translation also makes use of a number of derived forms of kindsΓconstructorsΓand expressions. These are shown in Figure 6.

The representation of tuples as records with numbered fields is copied from SML. The encoding of multi-argument functions as single-argument functions is also very standardΓas is the encoding of booleans as a sum type.

The purpose of the **catch** form is to serve as a special handler for the "fail" exceptionΓ and to propagate all other exceptions. The fail exception is only used by the elaborator to signal a failure in pattern-matching.

$knd_1 \times \cdots \times knd_n$	$\mapsto \{1{:}knd_1, \ldots, n{:}knd_n\}$
knd^n	$\mapsto \{1{:}knd, \ldots, n{:}knd\}$
Unit	$\mapsto \{\}$
$\mathsf{Bool}_{\langle lab \rangle}$	$\mapsto \Sigma_{\langle lab \rangle}\ (\overline{1}{\mapsto}\mathsf{Unit}, \overline{2}{\mapsto}\mathsf{Unit})$
$con_1 \times \cdots \times con_n$	$\mapsto \{\overline{1}{=}con_1, \cdots, \overline{\mathrm{n}}{=}con_n\}$
$\lambda(var_1, \ldots, var_n).con$	$\mapsto \lambda var{:}\Omega^n.(\{\pi_1\ var/var_1\} \cdots \{\pi_n\ var/var_n\} con)$
$(con_1, \ldots, con_n)$	$\mapsto \{1{=}con_1, \ldots, n{=}con_n\}$
$(exp_1, \cdots, exp_n)$	$\mapsto \{\overline{1}{=}exp_1, \cdots, \overline{\mathrm{n}}{=}exp_n\}$
$\lambda(var{:}con){:}con'.exp$	$\mapsto \pi_{\overline{1}}\ \mathsf{fix}\ var'(var{:}con){:}con'{\mapsto}exp\ \mathsf{end}$ $var' \notin \mathrm{FV}(exp)$
$\lambda(var_1{:}con_1, \ldots, var_n{:}con_n){:}con.exp$	$\mapsto \lambda(var{:}con_1 \times \cdots \times con_n){:}con.$ $\{\pi_1\ var/var_1\} \cdots \{\pi_n\ var/var_n\} exp$ $var \notin \mathrm{FV}(exp)$
$\mathsf{let}\ bnd_1, \ldots, bnd_n\ \mathsf{in}\ exp\ \mathsf{end}$	$\mapsto [1{=}bnd_1, \ldots, n{=}bnd_n, (n+1){=}exp].(n+1)$
fail^{con}	$\mapsto \mathsf{raise}^{con}\ basis.\mathrm{fail}^*.\mathrm{fail}$
$\mathsf{catch}^{con}\ exp\ \mathsf{with}\ exp'$	$\mapsto \mathsf{handle}\ exp\ \mathsf{with}\ (\lambda var{:}\mathsf{Tagged}.$ $\mathsf{iftagof}\ var\ \mathsf{is}\ basis.\mathrm{fail}^*.\mathrm{tag}\ \mathsf{then}\ \lambda var{:}\mathsf{Unit}.exp'\ \mathsf{else}\ \mathsf{raise}^{con}\ va$ $var \notin \mathrm{FV}(exp')$
$\mathsf{pproj}^{\Sigma(con_1, \ldots, con_n)}_{lab_i}\ (exp, exp')$	$\mapsto \mathsf{case}^{con'}\ exp\ \mathsf{of}$ $\lambda var{:}con'_{lab_1}.\mathsf{raise}^{con_i}\ exp', \ldots,$ $\lambda var{:}con'_{lab_i}.\mathsf{proj}^{con'_{lab_i}}_{lab_i}\ var, \ldots$ $\lambda var{:}con'_{lab_n}.\mathsf{raise}^{con_i}\ exp'\ \mathsf{end}$ where $con'_{\langle lab \rangle} = \Sigma_{\langle lab \rangle}\ (lab_1{\mapsto}con_1, \ldots, lab_n{\mapsto}con_n)$
false	$\mapsto \mathsf{inj}^{\mathsf{Bool}}_{\overline{1}}\ \{\}$
true	$\mapsto \mathsf{inj}^{\mathsf{Bool}}_{\overline{2}}\ \{\}$
$\mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ exp_3$	$\mapsto \mathsf{case}^{\mathsf{Bool}}\ exp_1\ \mathsf{of}\ \lambda var{:}\mathsf{Bool}_{\overline{1}}.exp_3, \lambda var{:}\mathsf{Bool}_{\overline{2}}.exp_2\ \mathsf{end}$ $var \notin \mathrm{FV}(exp_2, exp3)$
$exp_1\ \mathsf{and}\ exp_2$	$\mapsto \mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ \mathsf{false}$
$\Lambda(var_1^{\langle \mathrm{eq} \rangle_1}, \ldots, var_n^{\langle \mathrm{eq} \rangle_n}).exp$	$\mapsto \lambda(var{:}[1^*{:}[1{\rhd}var_1{:}\Omega\langle, \mathrm{eq}{:}var_1 \times var_1 \rightharpoonup \mathsf{Bool}\rangle_1], \ldots,$ $n^*{:}[n{\rhd}var_n{:}\Omega\langle, \mathrm{eq}{:}var_n \times var_n \rightharpoonup \mathsf{Bool}\rangle_n]]).$ $[\mathrm{it}{=}\{var.1^*.1/var_1\} \cdots \{var.n^*.n/var_n\} exp]$
$\forall(var_1^{\langle \mathrm{eq} \rangle_1}, \ldots, var_n^{\langle \mathrm{eq} \rangle_n}).con$	$\mapsto (var{:}[1^*{:}[1{\rhd}var_1{:}\Omega\langle, \mathrm{eq}{:}var_1 \times var_1 \rightharpoonup \mathsf{Bool}\rangle_1], \ldots,$ $n^*{:}[n{\rhd}var_n{:}\Omega\langle, \mathrm{eq}{:}var_n \times var_n \rightharpoonup \mathsf{Bool}\rangle_n]]) \rightarrow$ $[\mathrm{it}{:}\{var.1^*.1/var_1\} \cdots \{var.n^*.n/var_n\} con]$

Figure 6: Derived Forms

6.5 Judgment Forms

Judgment...	*Meaning...*
$\Gamma \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{con}$	expression
$\Gamma \vdash \mathit{match} \leadsto \mathit{exp} : \mathit{con}$	pattern match
$\Gamma \vdash \mathit{strdec} \leadsto \mathit{sbnds} : \mathit{sdecs}$	declaration
$\Gamma \vdash \mathit{strexp} \leadsto \mathit{mod} : \mathit{sig}$	structure expression
$\Gamma \vdash \mathit{spec} \leadsto \mathit{sdecs}$	signature specification
$\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig}$	signature expression
$\Gamma \vdash \mathit{ty} \leadsto \mathit{con} : \Omega$	type expression
$\Gamma \vdash \mathit{tybind} \leadsto \mathit{sbnds} : \mathit{sdecs}$	`type` definition
$\Gamma \vdash \mathit{datbind} \leadsto \mathit{sbnds} : \mathit{sdecs}$	`datatype` definition
$\Gamma \vdash_{\mathsf{ctx}} \mathit{labs} \leadsto \mathit{path} : \mathit{class}$	lookup in Γ
$\Gamma \vdash_{\mathsf{ctx}} \mathit{labs} \leadsto \mathit{path}$	
$\mathit{decs}; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{labs} \leadsto \mathit{labs}' : \mathit{class}$	lookup in signature
$\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}'$	
$\mathit{decs} \vdash_{\mathsf{inst}} \leadsto [\mathit{sbnds}_{\mathsf{v}}] : [\mathit{sdecs}']$	polymorphic instantiation
$\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{else}\ \mathit{exp} \leadsto \mathit{sbnds} : \mathit{sbnds}$	pattern compilation
$\mathit{decs} \vdash_{\mathsf{eq}} \mathit{con} \leadsto \mathit{exp}_{\mathsf{v}}$	equality compilation
$\mathit{decs} \vdash_{\mathsf{sub}} \mathit{path} : \mathit{sig}_0 \preceq \mathit{sig} \leadsto \mathit{mod} : \mathit{sig}'$	coercion compilation
$\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathsf{sub}} \mathit{sdec} \leadsto \mathit{sbnd} : \mathit{sdec}'$	
$\mathit{sig} \vdash_{\mathsf{wt}} \mathit{labs} := \mathit{con} : \mathit{knd} \leadsto \mathit{sig}' : \mathsf{Sig}$	impose definition
$\mathit{sig} \vdash_{\mathsf{sh}} \mathit{labs} := \mathit{labs}' : \mathit{knd} \leadsto \mathit{sig}' : \mathsf{Sig}$	impose sharing

6.6 Translation Rules

6.6.1 Expressions

$$\boxed{\Gamma \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{con}}$$

$$\frac{}{\Gamma \vdash \mathit{scon} \leadsto \mathit{scon} : \mathrm{type}(\mathit{scon})} \tag{175}$$

Rule 175: We assume a meta-level function type which gives the IL type of each constant.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ctx}} \overline{\mathit{longid}} \leadsto \mathit{path} : \mathit{con} \\ \text{Rule 177 does not apply.}\end{array}}{\Gamma \vdash \mathit{longid} \leadsto \mathit{path} : \mathit{con}} \tag{176}$$

Rule 176: Monomorphic variables.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : con \rightarrow con'}{\Gamma \vdash longid \rightsquigarrow \partial\,(path) : con \rightharpoonup con'} \tag{177}$$

Rule 176: Monomorphic datatype or exception constructors. Because the IL has no subsumption, these must be coerced to a partial function type when used as function values (not immediately applied).

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \rightarrow [\mathrm{it}{:}con] \\ \Gamma \vdash_{\mathrm{inst}} \rightsquigarrow mod : sig \\ \text{Rule 179 does not apply.}\end{array}}{\Gamma \vdash longid \rightsquigarrow path(mod).\mathrm{it} : con} \tag{178}$$

Rule 178: Instantiation of polymorphic variables. All polymorphic functions are translated into total functors whose body contains a single component with the label "it". The module *mod* is the structure of types (and equality functions, if needed) that are used to instantiate the polymorphic function.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \rightarrow [\mathrm{it}{:}con \rightarrow con'] \\ \Gamma \vdash_{\mathrm{inst}} \rightsquigarrow mod : sig\end{array}}{\Gamma \vdash longid \rightsquigarrow \partial\,(path(mod).\mathrm{it}) : con \rightharpoonup con'} \tag{179}$$

Rule 179: Instantitaion of polymorphic datatype constructors. As in Rule 176, we coerce to a partial function type.

$$\frac{\begin{array}{c}\sigma \text{ a permutation of } 1..n \\ var_1, \cdots, var_n \notin \mathrm{BV}(\Gamma) \\ lab_{\sigma(1)} < \cdots < lab_{\sigma(n)} \\ \Gamma \vdash expr_1 \rightsquigarrow exp_1 : con_1 \ \cdots \ \Gamma \vdash expr_n \rightsquigarrow exp_n : con_n\end{array}}{\begin{array}{l}\Gamma \vdash \ \{lab_1 \ \texttt{=} \ expr_1, \cdots, lab_n \ \texttt{=} \ expr_n\} \rightsquigarrow \\ \qquad \mathsf{let}\ var_1{=}exp_1, \ldots, var_n{=}exp_n\ \mathsf{in}\ \{\overline{lab_{\sigma(1)}}{=}var_{\sigma(1)}, \cdots, \overline{var_{\sigma(n)}}{=}var_{\sigma(n)}\}\ \mathsf{end} : \\ \qquad \{\overline{lab_{\sigma(1)}}{:}con_{\sigma(1)}, \cdots, \overline{lab_{\sigma(n)}}{:}con_{\sigma(n)}\}\end{array}} \tag{180}$$

Rule 180: The order in which labels appear in the record type is significant for the IL but not the EL, so in the translation we normalize record values and types by sorting the labels with some fixed ordering <. The order in which effects occur, however, is determined by the order of the expressions in the EL.

$$\frac{\begin{array}{c}\Gamma \vdash strdec \rightsquigarrow sbnds : sdecs \\ var \notin \mathrm{BV}(\Gamma) \qquad \Gamma, 1^{*}{\triangleright} var{:}[sdecs] \vdash expr \rightsquigarrow exp : con' \\ \Gamma, var{:}[sdecs] \vdash con' \equiv con : \Omega \qquad \Gamma \vdash con : \Omega\end{array}}{\Gamma \vdash \texttt{let}\ strdec\ \texttt{in}\ expr\ \mathsf{end} \rightsquigarrow \mathsf{let}\ var{=}[mod]\ \mathsf{in}\ exp\ \mathsf{end} : con} \tag{181}$$

Rule 181: The "starred label" convention is used here to make the locally-defined bindings accessible while translating *expr*. The elaborator verifies that the translated expression can be given a a type, which must not depend on any abstract types (e.g., datatypes) defined in *strdec*.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}'' \rightharpoonup \mathit{con} \\ \Gamma \vdash \mathit{expr}' \rightsquigarrow \mathit{exp}' : \mathit{con}' \\ \Gamma \vdash \mathit{con}' \equiv \mathit{con}'' : \Omega\end{array}}{\Gamma \vdash \mathit{expr}\ \ \mathit{expr}' \rightsquigarrow \mathit{exp}\ \mathit{exp}' : \mathit{con}} \tag{182}$$

Rule 182: General application.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \mathit{longid} \rightsquigarrow \mathit{path} : \mathit{con}' \rightarrow \mathit{con} \\ \Gamma \vdash \mathit{expr}' \rightsquigarrow \mathit{exp}' : \mathit{con}'\end{array}}{\Gamma \vdash \mathit{longid}\ \ \mathit{expr}' \rightsquigarrow \mathit{exp}\ \mathit{exp}' : \mathit{con}} \tag{183}$$

Rule 183: Application of a monomorphic datatype constructor, which is valuable if *expr'* is valuable.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \mathit{longid} \rightsquigarrow \mathit{path} : \mathit{sig} \rightarrow [\mathrm{it}{:}\mathit{con}' \rightarrow \mathit{con}] \\ \Gamma \vdash_{\mathrm{inst}} \rightsquigarrow \mathit{mod} : \mathit{sig} \\ \Gamma \vdash \mathit{expr}' \rightsquigarrow \mathit{exp}' : \mathit{con}'\end{array}}{\Gamma \vdash \mathit{longid}\ \ \mathit{expr}' \rightsquigarrow (\mathit{path}\,(\mathit{mod}).\mathrm{it})\ \mathit{exp}' : \mathit{con}} \tag{184}$$

Rule 184: Application of a polymorphic datatype constructor, which is valuable if *expr'* is valuable.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \\ \Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con}' : \Omega \qquad \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \Omega\end{array}}{\Gamma \vdash \mathit{expr}\ :\ \mathit{ty} \rightsquigarrow \mathit{exp} : \mathit{con}} \tag{185}$$

Rule 185: Type constraints on expressions are verified, but do not appear in the translation.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \\ \Gamma \vdash \mathit{match} \rightsquigarrow \mathit{exp}' : \mathsf{Tagged} \rightharpoonup \mathit{con}' \\ \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \Omega \\ \mathit{var} \notin \mathrm{BV}(\Gamma)\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{expr}\ \texttt{handle}\ \mathit{match} \rightsquigarrow \\ \mathsf{handle}\ \mathit{exp}\ \mathsf{with} \\ \quad \lambda(\mathit{var}{:}\mathsf{Tagged}){:}\mathit{con}.(\mathsf{catch}^{\mathit{con}}\ \mathit{exp}'\ \mathit{var}\ \mathsf{with}\ \mathsf{raise}^{\mathit{con}}\ \mathit{var}) : \\ \mathit{con}\end{array}} \tag{186}$$

Rule 186: The handling expression *exp' var* will fail if the handler pattern does not match the exception caught by the IL handle, in which case we re-raise the exception.

$$\frac{\Gamma \vdash \mathit{expr} \leadsto \mathit{exp} : \mathsf{Tagged} \qquad \Gamma \vdash \mathit{con} : \Omega}{\Gamma \vdash \texttt{raise}\ \mathit{expr} \leadsto \mathsf{raise}^{\mathit{con}}\ \mathit{exp} : \mathit{con}} \tag{187}$$

Rule 187: `raise` expressions can be given any (valid) type. To preserve the property that every IL expression has a unique type up to equivalence, we annotate the IL raise with this type.

$$\frac{\Gamma \vdash \mathit{match} \leadsto \mathit{exp} : \mathit{con}_1 \rightharpoonup \mathit{con}_2 \qquad \mathit{var} \notin \mathrm{BV}(\Gamma)}{\begin{array}{l}\Gamma \vdash \texttt{fn}\ \mathit{match} \leadsto \\ \qquad \lambda(\mathit{var}{:}\mathit{con}_1){:}\mathit{con}_2.(\mathsf{catch}^{\mathit{con}_2}\ \mathit{exp}\ \mathit{var}\ \mathsf{with}\ \mathsf{raise}^{\mathit{con}_2}\ \mathit{basis}.\overline{\mathtt{Match}}^{*}.\overline{\mathtt{Match}}) : \\ \qquad \mathit{con}_1 \rightharpoonup \mathit{con}_2\end{array}} \tag{188}$$

Rule 188: The application *exp var* will fail if the match fails; we turn the failure into a match exception. The resulting function has a partial type because it can (syntactically) raise an exception.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr}_1 \leadsto \mathit{exp}_1 : \mathit{con}_1 \qquad \Gamma \vdash \mathit{expr}_2 \leadsto \mathit{exp}_2 : \mathit{con}_2 \\ \Gamma \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \Omega \qquad \Gamma \vdash_{\mathsf{eq}} \mathit{con}_1 \leadsto \mathit{exp}\end{array}}{\Gamma \vdash \mathit{expr}_1\ \texttt{=}\ \mathit{expr}_2 \leadsto \mathit{exp}\,(\mathit{exp}_1, \mathit{exp}_2) : \mathsf{Bool}} \tag{189}$$

Rule 189: Translation of equality comparison; *exp* is the equality function generated by the equality compiler and has type $\mathit{con} \times \mathit{con} \rightharpoonup \mathsf{Bool}$.

6.6.2 Matches

$$\boxed{\Gamma \vdash \mathit{match} \leadsto \mathit{exp} : \mathit{con}}$$

$$\frac{\begin{array}{c}\mathit{var}, \mathit{var}' \notin \mathrm{BV}(\Gamma) \qquad \Gamma \vdash \mathit{con}' : \Omega \\ \Gamma \vdash \mathit{pat} \Leftarrow \mathit{var}' : \mathit{con}'\ \mathsf{else}\ \mathit{basis}.\mathrm{fail}^{*}.\mathrm{fail} \leadsto \mathit{sbnds} : \mathit{sdecs} \\ \Gamma, 1^{*} \triangleright \mathit{var}{:}[\mathit{sdecs}] \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{con}\end{array}}{\Gamma \vdash \mathit{pat}\ \texttt{=>}\ \mathit{expr} \leadsto \lambda(\mathit{var}'{:}\mathit{con}'){:}\mathit{con}.\mathsf{let}\ \mathit{var}{=}[\mathit{sbnds}]\ \mathsf{in}\ \mathit{exp}\ \mathsf{end} : \mathit{con}' \rightharpoonup \mathit{con}} \tag{190}$$

Rule 190: The result of translating a match is a function that may fail if the match fails. The IL let expression is well-formed because the pattern compiler returns no type components in *sbnds*.

$$\frac{\begin{array}{c}\mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{mrule} \leadsto \mathit{exp} : \mathit{con}_1 \rightharpoonup \mathit{con}_2 \\ \Gamma \vdash \mathit{match} \leadsto \mathit{exp}' : \mathit{con}_1' \rightharpoonup \mathit{con}_2' \\ \Gamma \vdash \mathit{con}_1 \rightharpoonup \mathit{con}_2 \equiv \mathit{con}_1' \rightharpoonup \mathit{con}_2' : \Omega\end{array}}{\Gamma \vdash \mathit{mrule}\ \texttt{|}\ \mathit{match} \leadsto \lambda(\mathit{var}{:}\mathit{con}_1){:}\mathit{con}_2.\mathsf{catch}^{\mathit{con}}\ \mathit{exp}\ \mathit{var}\ \mathsf{with}\ \mathit{exp}'\ \mathit{var} : \mathit{con}' \rightharpoonup \mathit{con}} \tag{191}$$

Rule 191: The failure of pattern matching in the first clause is caught, and we try again with the next clause.

6.6.3 Declarations

$$\boxed{\Gamma \vdash \mathit{strdec} \leadsto \mathit{sbnds} : \mathit{sdecs}}$$

$$\frac{\begin{array}{c} \mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{con} \\ \Gamma, \mathit{var}{:}\mathit{con} \vdash \mathit{pat} \Leftarrow \mathit{var} : \mathit{con} \text{ else } \mathit{basis}.\overline{\mathtt{Bind}}^{*}.\overline{\mathtt{Bind}} \leadsto \mathit{sbnds} : \mathit{sdecs} \end{array}}{\Gamma \vdash \mathtt{val}\ ()\ \mathit{pat} \mathtt{\ =\ } \mathit{expr} \leadsto 1{\triangleright}\mathit{var}{=}\mathit{exp}, \mathit{sbnds} : 1{\triangleright}\mathit{var}{:}\mathit{con}, \mathit{sdecs}} \qquad (192)$$

Rule 192: Monomorphic, non-recursive variable bindings.

$$\frac{\begin{array}{l} \mathit{sig}_p = [\overline{\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1}^{*}{:}[\overline{\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1}{\triangleright}\mathit{var}{:}\Omega,\langle, \mathrm{eq}{:}\mathit{var}\times\mathit{var}\rightharpoonup\mathsf{Bool}\rangle_1], \cdots, \\ \qquad \overline{\langle \mathit{eq}\rangle_n \mathit{tyvar}_n}^{*}{:}[\overline{\langle \mathit{eq}\rangle_n \mathit{tyvar}_n}{\triangleright}\mathit{var}{:}\Omega,\langle, \mathrm{eq}{:}\mathit{var}\times\mathit{var}\rightharpoonup\mathsf{Bool}\rangle_n], \\ \qquad 1^{*}{:}[1{\triangleright}\mathit{var}{:}\Omega\langle, \mathrm{eq}{:}\mathit{var}\times\mathit{var}\rightharpoonup\mathsf{Bool}\rangle_{1'}], \cdots, \\ \qquad m^{*}{:}[m{\triangleright}\mathit{var}{:}\Omega\langle, \mathrm{eq}{:}\mathit{var}\times\mathit{var}\rightharpoonup\mathsf{Bool}\rangle_{m'}]] \\ \Gamma, 1^{*}{\triangleright}\mathit{var}_p{:}\mathit{sig}_p \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{con} \\ \Gamma, 1^{*}{\triangleright}\mathit{var}_p{:}\mathit{sig}_p \vdash \mathit{exp} \downarrow \mathit{con} \\ \Gamma, 1^{*}{\triangleright}\mathit{var}_p{:}\mathit{sig}_p \vdash \mathit{basis}.\overline{\mathtt{Bind}}^{*}.\overline{\mathtt{Bind}} \Leftarrow \mathit{exp} : \mathit{con} \text{ else } \mathit{pat} \leadsto \mathit{sbnd}_1, \ldots, \mathit{sbnd}_n : \mathit{sdec}_1, \ldots, \mathit{sdec}_n \\ \forall i \in 1..n : \\ \qquad \mathit{sbnd}_i = \mathit{lab}_i{=}\mathit{exp}_i \\ \qquad \mathit{sdec}_i = \mathit{lab}_i{:}\mathit{con}_i \\ \qquad \mathit{sbnd}'_i = \begin{cases} \mathit{lab}_i{=}(\mathit{var}_p{:}\mathit{sig}_p){\to}[\mathrm{it}{=}\mathit{exp}_i] & \text{if } \Gamma, \mathit{var}_p{:}\mathit{sig}_p \vdash \mathit{sbnd}_i \downarrow \mathit{sdec}_i \\ \mathit{lab}_i{=}\mathit{exp}_i & \text{if } \Gamma \vdash \mathit{exp}_i : \mathit{con}_i \end{cases} \\ \qquad \mathit{sdec}'_i = \begin{cases} \mathit{lab}_i{:}(\mathit{var}_p{:}\mathit{sig}_p){\to}[\mathrm{it}{:}\mathit{con}_i] & \text{if } \Gamma, \mathit{var}_p{:}\mathit{sig}_p \vdash \mathit{sbnd}_i \downarrow \mathit{sdec}_i \\ \mathit{lab}_i{:}\mathit{con}_i & \text{if } \Gamma \vdash \mathit{exp}_i : \mathit{con}_i \end{cases} \end{array}}{\begin{array}{c} \Gamma \vdash \mathtt{val}\ (\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1, \cdots, \langle \mathit{eq}\rangle_n \mathit{tyvar}_n)\ \mathit{pat} \mathtt{\ =\ } \mathit{expr} \leadsto \\ \mathit{sbnd}'_1, \ldots, \mathit{sbnd}'_n : \mathit{sdec}'_1, \ldots, \mathit{sdec}'_n \end{array}}$$

$$(193)$$

Rule 193: Polymorphic, non-recursive `val` bindings. We assume a prepass which annotates `val` declarations with the explicit type variables implicitly scoped by that declaration. Type inference may introduce m new type (or equality type) variables which not mentioned in the source (as in `val f = fn x => x` or `val f = fn x => (x=x)`). In this formulation, some variables in the pattern may become polymorphic while others may remain monomorphic. Therefore, for each i, either sbnd'_i and sdec'_i must both choose the first option (polymorphic) or must both choose the second option (monomorphic).

$$\frac{\begin{array}{l}
sig_p = [\overline{\langle eq\rangle_1 tyvar_1}^{*}{:}[\overline{\langle eq\rangle_1 tyvar_1}\triangleright var{:}\Omega\langle, \mathrm{eq}{:}var\times var\rightharpoonup \mathsf{Bool}\rangle_1], \cdots,\\
\qquad \overline{\langle eq\rangle_m tyvar_m}^{*}{:}[\overline{\langle eq\rangle_m tyvar_m}\triangleright var{:}\Omega\langle,\mathrm{eq}{:}var\times var\rightharpoonup\mathsf{Bool}\rangle_m]\\
\qquad 1^*{:}[1\triangleright var{:}\Omega\langle,\mathrm{eq}{:}var\times var\rightharpoonup\mathsf{Bool}\rangle_{1'}],\cdots,\\
\qquad k^*{:}[k\triangleright var{:}\Omega\langle,\mathrm{eq}{:}var\times var\rightharpoonup\mathsf{Bool}\rangle_{k'}]]\\
\quad \Gamma' = \Gamma, lab^*\triangleright var_p{:}sig_p, \overline{id_1}\triangleright var'_1{:}con_1\rightharpoonup con'_1,\cdots,\overline{id_n}\triangleright var'_n{:}con_n\rightharpoonup con'_n\\
\forall i\in 1..n:\\
\quad \Gamma'\vdash match_i \leadsto \lambda(var_i{:}con_i){:}con'_i.exp_i : con_i\rightharpoonup con'_i\\
\text{If } \Gamma\vdash_{\mathrm{ctx}} id_i \leadsto path'' : con'' \text{ then } path''.\mathrm{expose}, path''.\mathrm{tag} \text{ aren't well-formed.}\\
exp = \mathsf{fix}\, var'_1(var_1{:}con_1){:}con'_1\mapsto exp_1,\cdots, var'_m(var_m{:}con_m){:}con'_m\mapsto exp_m\ \mathsf{end}
\end{array}}{\begin{array}{l}
\Gamma\vdash \texttt{val}\ (\langle eq\rangle_1 tyvar_1,\cdots,\langle eq\rangle_m tyvar_m)\ \texttt{rec}\ \{(lab_i\texttt{=}id_i)_{i=1}^n\}\ \texttt{=}\ \{(lab_i\texttt{=fn}\ match_i)_{i=1}^n\}\leadsto\\
\quad \overline{id_1}=\lambda var_p{:}sig_p.[\mathrm{it}=\pi_{\overline{1}}\, exp],\ldots,\overline{id_n}=\lambda var_p{:}sig_p.[\mathrm{it}=\pi_{\overline{n}}\, exp]:\\
\quad \overline{id_1}{:}(var_p{:}sig_p)\to[\mathrm{it}{:}con_1\rightharpoonup con'_1],\ldots,\overline{id_n}{:}(var_p{:}sig_p)\to[\mathrm{it}{:}con_n\rightharpoonup con'_n]
\end{array}} \tag{194}$$

Rule 194: This rule handles recursive `val` bindings. As in Rule 193, we assume that the implicit scoping of explicit type variables has been made explicit by a prepass over the EL. We also assume that `val rec ... and ...` is syntactic sugar for a single `val rec` binding of a record of functions. Since this rule does not use the pattern compiler, we must explicitly check that we're do not redefine datatype or exception constructors.

$$\frac{\begin{array}{c}
\Gamma\vdash strdec_1 \leadsto sbnds_1 : sdecs_1\\
\Gamma, sdecs_1\vdash strdec_2\leadsto sbnds_2 : sdecs_2
\end{array}}{\Gamma\vdash strdec_1\ strdec_2\leadsto sbnds_1{+}{+}sbnds_2 : sdecs_1{+}{+}sdecs_2} \tag{195}$$

Rule 195: We use the syntactic-concatenation-with-renaming operation defined in Section 6.2.

$$\frac{\forall i\in 1..n:\quad \Gamma\vdash_{\mathrm{ctx}} \overline{longstrid_i}\leadsto path_i : sig_i}{\begin{array}{l}
\Gamma\vdash \mathsf{open}\ longstrid_1\ \cdots\ longstrid_n\leadsto\\
\qquad 1^*{=}path_1,\cdots,n^*{=}path_n : 1^*{:}sig_1,\cdots,n^*{:}sig_n
\end{array}} \tag{196}$$

$$\frac{\Gamma\vdash ty\leadsto con:\Omega \qquad var\notin \mathrm{BV}(\Gamma)}{\begin{array}{l}
\Gamma\vdash \texttt{exception}\ id\ \texttt{of}\ ty\leadsto\\
\qquad \overline{id}^{*}=[\mathrm{tag}\triangleright var=\mathsf{new\_tag}[con], \overline{id}=\lambda(var'{:}con){:}\mathsf{Tagged}.\mathsf{tag}(var,var')]:\\
\qquad \overline{id}^{*}{:}[\mathrm{tag}\triangleright var{:}con\ \mathsf{Tag}, \overline{id}{:}con\to\mathsf{Tagged}]
\end{array}} \tag{197}$$

$$\frac{}{\begin{array}{l}
\Gamma\vdash \texttt{exception}\ id\leadsto\\
\qquad \overline{id}^{*}=[\mathrm{tag}\triangleright var=\mathsf{new\_tag}[\mathsf{Unit}], \overline{id}=\mathsf{tag}(var,\{\})]:\\
\qquad \overline{id}^{*}{:}[\mathrm{tag}\triangleright var{:}\mathsf{Unit}\ \mathsf{Tag}, \overline{id}{:}\mathsf{Tagged}]
\end{array}} \tag{198}$$

$$\frac{\begin{array}{c}
\Gamma\vdash_{\mathrm{ctx}} \overline{longid}\leadsto path.lab : con\\
\Gamma\vdash path.\mathrm{tag} : con'
\end{array}}{\begin{array}{l}
\Gamma\vdash \texttt{exception}\ id\ \texttt{=}\ longid\leadsto\\
\qquad \overline{id}^{*}=[\mathrm{tag}=path.\mathrm{tag}, \overline{id}=path.lab] : \overline{id}^{*}{:}[\mathrm{tag}{:}con', \overline{id}{:}con]
\end{array}} \tag{199}$$

Rule 199: Structures containing a "tag" component are created by EL exception declrations only.

$$\frac{\begin{array}{c} \mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{strdec}_1 \rightsquigarrow \mathit{sbnds}_1 : \mathit{sdecs}_1 \\ \Gamma, 1^{*}\triangleright \mathit{var}{:}[\mathit{sdecs}_1] \vdash \mathit{strdec}' \rightsquigarrow \mathit{sbnds}_2 : \mathit{sdecs}_2 \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{local}\ \mathit{strdec}\ \texttt{in}\ \mathit{strdec}'\ \texttt{end} \rightsquigarrow \\ \quad 1\triangleright \mathit{var}{=}[\mathit{sbnds}_1], \mathit{sbnds}_2 : 1\triangleright \mathit{var}{:}[\mathit{sdecs}_1], \mathit{sdecs}_2 \end{array}} \tag{200}$$

Rule 200: We create bindings for all of the declarations, but the local bindings are segregated into a substructure inaccessible from the EL.

$$\frac{\Gamma \vdash \mathit{tybind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}{\Gamma \vdash \texttt{type}\ \mathit{tybind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}} \tag{201}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longtycon}} \rightsquigarrow \mathit{path}.\overline{\mathit{tycon}} : \Omega^m{\Rightarrow}\Omega \\ \Gamma \vdash \mathit{path} : [\overline{\mathit{tycon}}\triangleright \mathit{var}{:}\Omega^m{\Rightarrow}\Omega, \mathit{lab}_1\triangleright \mathit{dec}_1, \cdots, \mathit{lab}_n\triangleright \mathit{dec}_n] \\ \mathit{lab}_n = \mathrm{expose} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{datatype}\ \texttt{(}\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_m\texttt{)}\ \mathit{tycon}' \qquad\qquad \rightsquigarrow \\ \qquad \texttt{= datatype}\ \texttt{(}\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_m\texttt{)}\ \mathit{longtycon} \\ \qquad \overline{\mathit{tycon}'}^{*}{=}[\overline{\mathit{tycon}'}\triangleright \mathit{var}{=}\mathit{path}.\overline{\mathit{tycon}}, \mathit{lab}_1{=}\mathit{path}.\mathit{lab}_1, \cdots, \mathit{lab}_n{=}\mathit{path}.\mathit{lab}_n] : \\ \qquad \overline{\mathit{tycon}'}^{*}{=}[\overline{\mathit{tycon}'}\triangleright \mathit{var}{:}\Omega^m{\Rightarrow}\Omega{=}\mathit{path}.\overline{\mathit{tycon}}, \mathit{lab}_1{=}\mathit{dec}_1, \cdots, \mathit{lab}_n{=}\mathit{dec}_n] \end{array}} \tag{202}$$

Rule 202: SML '96 adds the ability to copy datatypes. Only structures which are the translations of EL datatype declarations have an initial type component and a final component named "expose".

$$\frac{\Gamma \vdash \mathit{datbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}{\Gamma \vdash \texttt{datatype}\ \mathit{datbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}} \tag{203}$$

$$\frac{\Gamma \vdash \mathit{strbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}{\Gamma \vdash \texttt{structure}\ \mathit{strbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}} \tag{204}$$

$$\frac{\Gamma \vdash \mathit{funbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}{\Gamma \vdash \texttt{functor}\ \mathit{funbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}} \tag{205}$$

6.6.4 Structure Expressions

$$\boxed{\Gamma \vdash \mathit{strexp} \rightsquigarrow \mathit{mod} : \mathit{sig}}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longstrid}} \rightsquigarrow \mathit{path} : \mathit{sig}}{\Gamma \vdash \mathit{longstrid} \rightsquigarrow \mathit{path} : \mathit{sig}} \tag{206}$$

$$\frac{\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}{\Gamma \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}} \tag{207}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longfunid}} \leadsto \mathit{path}_f : (\mathit{var}_1{:}\mathit{sig}_1) \rightharpoonup \mathit{sig}_2 \\ \Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longstrid}} \leadsto \mathit{path} : \mathit{sig} \\ \Gamma \vdash_{\mathrm{sub}} \mathit{path} : \mathit{sig} \preceq \mathit{sig}_1 \leadsto \mathit{mod} : \mathit{sig}' \\ \Gamma \vdash (\mathit{var}_1{:}\mathit{sig}) \rightharpoonup \mathit{sig}_2 \equiv \mathit{sig}' \rightharpoonup \mathit{sig}'' : \mathsf{Sig}\end{array}}{\Gamma \vdash \mathit{longfunid}\,(\mathit{longstrid}) \leadsto (\mathit{path}_f{:}\mathit{sig}' \rightharpoonup \mathit{sig}'')\; \mathit{mod} : \mathit{sig}''} \tag{208}$$

Rule 208: We insert an explicit coercion to drop and reorder components of the argument structure (which has signature *sig*), in order to match the domain signature of the functor (sig_1). The signature sig' is the most-specific (and fully transparent) signature of the coerced structure, which may expose more types (is a sub-signature of) sig_1.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \mathit{longstrid} \leadsto \mathit{path} : \mathit{sig} \qquad \Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig}' : \mathsf{Sig} \\ \Gamma \vdash_{\mathrm{sub}} \mathit{path} : \mathit{sig} \preceq \mathit{sig}' \leadsto \mathit{mod} : \mathit{sig}''\end{array}}{\Gamma \vdash \mathit{longstrid} \;:\; \mathit{sigexp} \leadsto \mathit{mod} : \mathit{sig}''} \tag{209}$$

Rule 209: As in SML, ascribing a signature to a structure using ":" hides components (this hiding being accomplished here via an explicit coercion), but allows the identity of the remaining type components to leak through. The rules for coercions ensure that sig'' will be fully transparent, maximizing propagation of type information.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \mathit{longstrid} \leadsto \mathit{path} : \mathit{sig} \qquad \Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig}' : \mathsf{Sig} \\ \Gamma \vdash_{\mathrm{sub}} \mathit{path} : \mathit{sig} \preceq \mathit{sig}' \leadsto \mathit{mod} : \mathit{sig}''\end{array}}{\Gamma \vdash \mathit{longstrid} \;:\; \mathit{sigexp} \leadsto (\mathit{mod}{:}\mathit{sig}') : \mathit{sig}'} \tag{210}$$

Rule 210: Ascribing a signature to a structure with :> not only hides components, but restricts information about types to that which appears in the signature.

$$\frac{\begin{array}{c}\mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{strdec} \leadsto \mathit{sbnds} : \mathit{sdecs} \\ \Gamma, 1^{*}\triangleright \mathit{var}{:}[\mathit{sdecs}] \vdash \mathit{strexp} \leadsto \mathit{mod} : \mathit{sig}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ \mathit{strdec}\ \mathtt{in}\ \mathit{strexp}\ \mathtt{end} \leadsto \\ \qquad [1\triangleright \mathit{var}{=}[\mathit{sbnds}], 2^{*}{=}\mathit{mod}] : [1\triangleright \mathit{var}{:}[\mathit{sdecs}], 2^{*}{:}\mathit{sig}]\end{array}} \tag{211}$$

6.6.5 Structure Bindings

$$\boxed{\Gamma \vdash \mathit{strbind} \leadsto \mathit{sbnds} : \mathit{sdecs}}$$

$$\frac{\Gamma \vdash \mathit{strexp}_1 \leadsto \mathit{mod}_1 : \mathit{sig}_1 \qquad \langle \cdots \qquad \Gamma \vdash \mathit{strexp}_n \leadsto \mathit{mod}_n : \mathit{sig}_n \rangle}{\begin{array}{l}\Gamma \vdash \mathit{strid}_1 \texttt{ = } \mathit{strexp}_1\ \langle \mathsf{and}\ \cdots\ \mathsf{and}\ \mathit{strid}_n \texttt{ = } \mathit{strexp}_n \rangle \leadsto \\ \qquad \overline{\mathit{strid}_1}{=}\mathit{mod}_1 \langle, \cdots, \overline{\mathit{strid}_n}{=}\mathit{mod}_n \rangle : \overline{\mathit{strid}_1}{:}\mathit{sig}_1 \langle, \cdots, \overline{\mathit{strid}_n}{:}\mathit{sig}_n \rangle\end{array}} \tag{212}$$

6.6.6 Functor Bindings

$$\boxed{\Gamma \vdash \mathit{funbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}$$

$$\frac{\begin{array}{cc} & \mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{sigexp}_1 \rightsquigarrow \mathit{sig}_1 : \mathsf{Sig} & \Gamma, \overline{\mathit{strid}_1} \triangleright \mathit{var}{:}\mathit{sig}_1 \vdash \mathit{strexp}_1 \rightsquigarrow \mathit{mod}_1 : \mathit{sig}'_1 \\ & \langle \vdots \rangle \\ \langle \Gamma \vdash \mathit{sigexp}_n \rightsquigarrow \mathit{sig}_n : \mathsf{Sig} & \Gamma, \overline{\mathit{strid}_n} \triangleright \mathit{var}{:}\mathit{sig}_n \vdash \mathit{strexp}_n \rightsquigarrow \mathit{mod}_n : \mathit{sig}'_n \rangle \end{array}}{\begin{array}{l} \Gamma \vdash \mathit{funid}_1 \texttt{(}\mathit{strid}_1 \texttt{ : } \mathit{sigexp}_1\texttt{) = } \mathit{strexp}_1 \\ \quad \langle \texttt{and} \ \cdots \ \texttt{and} \ \mathit{funid}_n \texttt{(}\mathit{strid}_n \texttt{ : } \mathit{sigexp}_1\texttt{) = } \mathit{strexp}_n \rangle \rightsquigarrow \\ \quad (\overline{\mathit{funid}_1} = \lambda \mathit{var}{:}\mathit{sig}_1.\mathit{mod}_1{:}\mathit{var}{:}\mathit{sig}_1 \rightharpoonup \mathit{sig}'_1)_{i=1}^{n} : \\ \quad (\overline{\mathit{funid}_1}{:}\mathit{var}{:}\mathit{sig}_1 \rightharpoonup \mathit{sig}'_1)_{i=1}^{n} \end{array}} \tag{213}$$

Rule 213: As for functions at the expression level, user-defined functors are given partial functor types.

6.6.7 Specifications

$$\boxed{\Gamma \vdash \mathit{spec} \rightsquigarrow \mathit{sdecs}}$$

$$\frac{}{\Gamma \vdash \cdot \rightsquigarrow \cdot} \tag{214}$$

$$\frac{\begin{array}{c} \mathrm{TV}(\mathit{ty}) = \emptyset \\ \Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega \end{array}}{\Gamma \vdash \texttt{val } \mathit{id} \texttt{ : } \mathit{ty} \rightsquigarrow \overline{\mathit{id}}{:}\mathit{con}} \tag{215}$$

Rule 215: A value specifiation is monomorphic if the EL type contains no type variables. (The set of type variables of the EL type *ty* is denoted by TV(*ty*).)

$$\frac{\begin{array}{c} \mathrm{TV}(\mathit{ty}) = \{\langle \mathit{eq} \rangle_1 \mathit{tyvar}_1, \cdots, \langle \mathit{eq} \rangle_n \mathit{tyvar}_n\} \neq \emptyset \\ \mathit{sig}_p = [\overline{\langle \mathit{eq} \rangle_1 \mathit{tyvar}_1}^{*}{:}[\overline{\langle \mathit{eq} \rangle_1 \mathit{tyvar}_1} \triangleright \mathit{var}{:}\Omega\langle, \mathrm{eq}{:}\mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_1], \cdots, \\ \overline{\langle \mathit{eq} \rangle_n \mathit{tyvar}_n}^{*}{:}[\overline{\langle \mathit{eq} \rangle_n \mathit{tyvar}_n} \triangleright \mathit{var}{:}\Omega\langle, \mathrm{eq}{:}\mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_n]] \\ \Gamma, \mathit{lab}^{*} \triangleright \mathit{var}{:}\mathit{sig}_p \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega \end{array}}{\Gamma \vdash \texttt{val } \mathit{id} \texttt{ : } \mathit{ty} \rightsquigarrow \overline{\mathit{id}}{:}(\mathit{var}{:}\mathit{sig}_p) \rightarrow [\mathrm{it}{:}\mathit{con}]} \tag{216}$$

Rule 216: Specification of a polymorphic value.

$$\frac{\Gamma \vdash \mathit{typdesc} \rightsquigarrow \mathit{sdecs}}{\Gamma \vdash \texttt{type } \mathit{typdesc} \rightsquigarrow \mathit{sdecs}} \tag{217}$$

$$\frac{\Gamma \vdash \mathit{etypdesc} \rightsquigarrow \mathit{sdecs}}{\Gamma \vdash \texttt{eqtype } \mathit{etypdesc} \rightsquigarrow \mathit{sdecs}} \tag{218}$$

$$\frac{\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega}{\Gamma \vdash \texttt{exception } \mathit{id} \texttt{ of } \mathit{ty} \rightsquigarrow \overline{\mathit{id}}{:}[\mathrm{tag}{:}\mathit{con}\ \mathsf{Tag}, \overline{\mathit{id}}{:}\mathit{con} \rightarrow \mathsf{Tagged}]} \tag{219}$$

$$\frac{}{\Gamma \vdash \texttt{exception}\ id \leadsto \overline{id}{:}[\mathrm{tag}{:}con\,\mathsf{Tag}, \overline{id}{:}\mathsf{Tagged}]} \tag{220}$$

$$\frac{\Gamma \vdash datbind \leadsto sbnd : 1^*{:}[sdecs]}{\Gamma \vdash \texttt{datatype}\ datbind \leadsto sdecs} \tag{221}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longtycon} \leadsto path.\overline{tycon} : \Omega^n{\Rightarrow}\Omega \\ \Gamma \vdash path : [\overline{tycon}{:}\Omega^n{\Rightarrow}\Omega, sdecs]\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{datatype}\ (tyvar_1\texttt{,}\cdots\texttt{,}tyvar_n)\ tycon' \qquad\qquad \leadsto \\ \qquad \texttt{=}\ \texttt{datatype}\ (tyvar_1\texttt{,}\cdots\texttt{,}tyvar_n)\ longtycon \\ \qquad \overline{tycon'}^{*}{:}[\overline{tycon'}{\triangleright}var{:}\Omega^n{\Rightarrow}\Omega{=}path.\overline{tycon}, sdecs]\end{array}} \tag{222}$$

$$\frac{\Gamma \vdash sigexp \leadsto sig : \mathsf{Sig}}{\Gamma \vdash \texttt{structure}\ strid\ \texttt{:}\ sigexp \leadsto \overline{strid}{:}sig} \tag{223}$$

$$\frac{\begin{array}{c}\Gamma \vdash sigexp \leadsto sig : \mathsf{Sig} \\ var \notin \mathrm{BV}(\Gamma) \\ \Gamma, \overline{strid}{\triangleright}var{:}sig \vdash sigexp' \leadsto sig' : \mathsf{Sig}\end{array}}{\Gamma \vdash \texttt{functor}\ funid\ \texttt{(}strid\ \texttt{:}\ sigexp\texttt{)}\ \texttt{:}\ sigexp' \leadsto \overline{funid}{:}(var{:}sig{\rightharpoonup}sig')} \tag{224}$$

$$\frac{\Gamma \vdash sigexp \leadsto [sdecs] : \mathsf{Sig}}{\Gamma \vdash \texttt{include}\ sigexp \leadsto sdecs} \tag{225}$$

$$\frac{\begin{array}{c}\Gamma \vdash spec_1 \leadsto sdecs_1 \qquad \Gamma, sdecs_1 \vdash spec_2 \leadsto sdecs_2 \\ \Gamma \vdash sdecs_1, sdecs_2\ \mathrm{ok}\end{array}}{\Gamma \vdash spec_1\ spec_2 \leadsto sdecs_1, sdecs_2} \tag{226}$$

Rule 226: We disallow redeclaring EL identifiers in a signature. In the presence of include, we cannot syntactically restrict the EL to guarantee the syntactic concatenation of $sdecs_1$ and $sdecs_2$ will be well-formed.

$$\frac{\begin{array}{c}\Gamma \vdash spec \leadsto sig : \mathsf{Sig} \\ var \notin \mathrm{BV}(\Gamma) \\ \Gamma; var{:}sig \vdash_{\mathsf{sig}} \overline{longid_1} \leadsto labs_1 : knd \\ \Gamma; var{:}sig \vdash_{\mathsf{sig}} \overline{longid_2} \leadsto labs_2 : knd \\ \Gamma, var{:}sig \vdash var.labs_1 \equiv var.labs_1' : knd \\ \Gamma, var{:}sig \vdash var.labs_2 \equiv var.labs_2' : knd \\ \left\{\begin{array}{c} sig \vdash_{\mathsf{sh}} labs_1' := labs_2' : knd \leadsto sig' : \mathsf{Sig} \\ \mathrm{or} \\ sig \vdash_{\mathsf{sh}} labs_2' := labs_1' : knd \leadsto sig' : \mathsf{Sig}\end{array}\right\}\end{array}}{\Gamma \vdash spec\ \texttt{sharing type}\ longid_1\ \texttt{=}\ longid_2 \leadsto sig' : \mathsf{Sig}} \tag{227}$$

Rule 227: A type component in a signature is considered abstract, and hence eligible to appear in a sharing constraint, if it is equivalent to an opaque type (type component that is not a type abbreviation) in the *spec*.

We find the two opaque types to which the given components are equivalent, and patch the signature such that the opaque type with the smaller scope becomes a type abbreviation for the other opaque type.

6.6.8 Signature Expressions

$$\boxed{\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig}}$$

$$\frac{\Gamma \vdash \mathit{spec} \leadsto \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \leadsto \mathit{sig} : \mathsf{Sig}} \tag{228}$$

$$\frac{\begin{array}{c} \mathit{var}, \mathit{var}_1, \cdots, \mathit{var}_n \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma, \overline{\mathit{tyvar}_1} \triangleright \mathit{var}_1{:}\Omega, \cdots, \overline{\mathit{tyvar}_n} \triangleright \mathit{var}_n{:}\Omega \vdash \mathit{ty} \leadsto \mathit{con} : \Omega \\ \Gamma; \mathit{var}{:}\mathit{sig} \vdash_{\mathrm{sig}} \overline{\mathit{longtycon}} \leadsto \mathit{labs} : \Omega^n \Rightarrow \Omega \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash \mathit{var}.\mathit{labs} \equiv \mathit{var}.\mathit{labs}' : \Omega^n \Rightarrow \Omega \\ \mathit{sig} \vdash_{\mathrm{wt}} \mathit{labs}' := \lambda(\mathit{var}_1, \cdots, \mathit{var}_n).\mathit{con} : \Omega^n \Rightarrow \Omega \leadsto \mathit{sig}' : \mathsf{Sig} \end{array}}{\Gamma \vdash \mathit{sigexp}\ \texttt{where type}\ \texttt{(}\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty} \leadsto \mathit{sig}' : \mathsf{Sig}} \tag{229}$$

Rule 229: Note that type EL type expression *ty* is evaluated using the ambient scope.

6.6.9 Type Expressions

$$\boxed{\Gamma \vdash \mathit{ty} \leadsto \mathit{con} : \Omega}$$

$$\frac{}{\Gamma \vdash \texttt{int} \leadsto \mathsf{Int} : \Omega} \tag{230}$$

Rule 230: There are analogous rules for the other base types (Float, Bool, Unit, String, and so on), and the base type constructors (`ref`).

$$\frac{}{\Gamma \vdash \texttt{exn} \leadsto \mathsf{Tagged} : \Omega} \tag{231}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{tyvar}} \leadsto \mathit{path} : \Omega}{\Gamma \vdash \mathit{tyvar} \leadsto \mathit{path} : \Omega} \tag{232}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathit{ty}_1 \leadsto \mathit{con}_1 : \Omega \qquad \cdots \qquad \Gamma \vdash \mathit{ty}_n \leadsto \mathit{con}_n : \Omega \\ \sigma \text{ a permutation of } 1..n \\ \mathit{lab}_{\sigma(1)} < \cdots < \mathit{lab}_{\sigma(n)} \end{array}}{\Gamma \vdash \texttt{\{}\mathit{lab}_1 \texttt{:} \mathit{ty}_1, \cdots, \mathit{lab}_n \texttt{:} \mathit{ty}_n\texttt{\}} \leadsto \{\overline{\mathit{lab}_{\sigma(1)}}{:}\mathit{con}_{\sigma(1)}, \cdots, \overline{\mathit{lab}_{\sigma(n)}}{:}\mathit{con}_{\sigma(n)}\} : \Omega} \tag{233}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longtycon}} \leadsto \mathit{path} : \Omega^n \Rightarrow \Omega \\ \forall i \in 1..n : \quad \Gamma \vdash \mathit{ty}_i \leadsto \mathit{con}_i : \Omega \end{array}}{\Gamma \vdash \texttt{(}\mathit{ty}_1\texttt{,} \cdots \texttt{,}\mathit{ty}_n\texttt{)}\ \mathit{longtycon} \leadsto \mathit{path}\,(\mathit{con}_1, \cdots, \mathit{con}_n) : \Omega} \tag{234}$$

$$\frac{\Gamma \vdash \mathit{ty} \leadsto \mathit{con} : \Omega \qquad \Gamma \vdash \mathit{ty}' \leadsto \mathit{con}' : \Omega}{\Gamma \vdash \mathit{ty}\ \texttt{->}\ \mathit{ty}' \leadsto \mathit{con} \rightharpoonup \mathit{con}' : \Omega} \tag{235}$$

Rule 235: There is no way to express total (→) types in the external language.

6.6.10 Type Definitions

$$\boxed{\Gamma \vdash \mathit{tybind} \leadsto \mathit{sbnds} : \mathit{sdecs}}$$

$$\begin{array}{c}
\mathit{var}_1, \cdots, \mathit{var}_n \notin \mathrm{BV}(\Gamma) \\
\Gamma, \overline{\mathit{tyvar}_1} \rhd \mathit{var}_1{:}\Omega, \cdots, \overline{\mathit{tyvar}_n} \rhd \mathit{var}_n{:}\Omega \vdash \mathit{ty} \leadsto \mathit{con}' : \Omega \\
\mathit{con} = \lambda(\mathit{var}_1, \cdots, \mathit{var}_n).\mathit{con}' \\
\langle \Gamma \vdash \mathit{tybind} \leadsto \mathit{sbnds} : \mathit{sdecs} \qquad \mathit{var} \notin \mathrm{FV}(\mathit{sdecs}) \rangle \\
\hline
\Gamma \vdash \texttt{(}\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{tybind} \rangle \leadsto \\
\quad \overline{\mathit{tycon}} \rhd \mathit{var} {=} \mathit{con} \langle, \mathit{sbnds} \rangle : \overline{\mathit{tycon}} \rhd \mathit{var}{:}\Omega^n \Rightarrow \Omega {=} \mathit{con} \langle, \mathit{sdecs} \rangle
\end{array} \tag{236}$$

6.6.11 Type Descriptions

$$\boxed{\Gamma \vdash \mathit{typdesc} \leadsto [\mathit{sdecs}]}$$

$$\begin{array}{c}
\langle \Gamma \vdash \mathit{typdesc} \leadsto \mathit{sdecs} \rangle \\
\hline
\Gamma \vdash \texttt{type}\ \texttt{(}\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n\texttt{)}\ \mathit{id}\ \langle\ \texttt{and}\ \mathit{typdesc} \rangle \leadsto \overline{\mathit{id}}{:}\Omega^n \Rightarrow \Omega \langle, \mathit{sdecs} \rangle
\end{array} \tag{237}$$

$$\begin{array}{c}
\mathit{var}_1, \cdots, \mathit{var}_n \notin \mathrm{BV}(\Gamma) \\
\Gamma, \overline{\mathit{tyvar}_1} \rhd \mathit{var}_1{:}\Omega, \cdots, \overline{\mathit{tyvar}_n} \rhd \mathit{var}_n{:}\Omega \vdash \mathit{ty} \leadsto \mathit{con}' : \Omega \\
\mathit{con} := \lambda(\mathit{var}_1, \cdots, \mathit{var}_n).\mathit{con}' \\
\langle \Gamma \vdash \mathit{typdesc} \leadsto \mathit{sdecs} \qquad \mathit{var} \notin \mathrm{FV} \mathit{sdecs} \rangle \\
\hline
\Gamma \vdash \texttt{type}\ \texttt{(}\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n\texttt{)}\ \mathit{id}\ \texttt{=}\ \mathit{ty}\ \langle\langle\ \texttt{and}\ \mathit{typdesc} \rangle\rangle \leadsto \overline{\mathit{id}}{:}\Omega^n \Rightarrow \Omega {=} \mathit{con} \langle, \mathit{sdecs} \rangle
\end{array} \tag{238}$$

6.6.12 Equality Type Descriptions

$$\boxed{\Gamma \vdash \mathit{etypdesc} \leadsto \mathit{sdecs}}$$

The polymorphic equality functions defined by this judgment the same as the default "structural" equality supplied by SML. EssentiallyΓa type is an equality type if and only if we can use this judgment to create the equality function at that type.

$$\begin{array}{c}
\mathit{sig}' = [\overline{\mathit{tyvar}_1}^{*}{:}[\overline{\mathit{tyvar}_1} \rhd \mathit{var}_1{:}\Omega, \mathrm{eq}{:}\mathit{var}_1 \times \mathit{var}_1 \rightharpoonup \mathsf{Bool}], \cdots, \\
\overline{\mathit{tyvar}_n}^{*}{:}[\overline{\mathit{tyvar}_n} \rhd \mathit{var}_n{:}\Omega, \mathrm{eq}{:}\mathit{var}_n \times \mathit{var}_n \rightharpoonup \mathsf{Bool}]] \\
\mathit{con} = \mathit{var}(\mathit{var}'.\overline{\mathit{tyvar}_1}^{*}.\overline{\mathit{tyvar}_1}, \cdots, \mathit{var}'.\overline{\mathit{tyvar}_n}^{*}.\overline{\mathit{tyvar}_n}) \\
\langle \Gamma \vdash \mathit{etypdesc} \leadsto \mathit{sdecs} \rangle \\
\hline
\Gamma \vdash \texttt{(}\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n\texttt{)}\ \mathit{id}\ \langle \texttt{and}\ \mathit{etypdesc} \rangle \leadsto \\
\quad \overline{\mathit{id}}^{*}{:}[\overline{\mathit{id}} \rhd \mathit{var}{:}\Omega^n \Rightarrow \Omega, \mathrm{eq}{:}(\mathit{var}'{:}\mathit{sig}') \rightarrow [\mathrm{it}{:}\mathit{con} \times \mathit{con} \rightharpoonup \mathsf{Bool}]] \langle, \mathit{sdecs} \rangle
\end{array} \tag{239}$$

6.6.13 Datatype Definitions

$$\boxed{\Gamma \vdash \mathit{datbind} \leadsto \mathit{sbnds} : \mathit{sdecs}}$$

Without loss of generalityΓwe may assume that there are no duplicate *tyvar*'s among the bindings.

$$
\frac{
\begin{array}{l}
\Gamma' := \Gamma, \overline{tycon_1}\triangleright var_1^{dt}{:}\Omega^{n_1}\Rightarrow\Omega \cdots, \overline{tycon_p}\triangleright var_p^{dt}{:}\Omega^{n_p}\Rightarrow\Omega \\
\Gamma'' := \Gamma', \overline{tyvar_{11}}\triangleright var_{11}{:}\Omega, \cdots, \overline{tyvar_{pn_p}}\triangleright var_{pn_p}{:}\Omega, \\
var_1, \ldots, var_n \notin \mathrm{BV}(\Gamma'') \\
\\
\Gamma'' \vdash \left\{\begin{array}{c}\texttt{unit}\\ \text{or}\\ ty_{ij}\end{array}\right\}_{ij} \rightsquigarrow con_{ij} : \Omega \\
\\
con_i^{\mathrm{sum}} := \Sigma\big(\overline{id_{i1}}\mapsto con_{i1}, \cdots, \overline{id_{im_1}}\mapsto con_{im_i}\big) \\
con_i' := \pi_i\,(\mu\,\lambda(var_1^{dt}, \ldots, var_p^{dt}).((\lambda(var_{j1}, \ldots, var_{jn_j}).\, con_j^{\mathrm{sum}})_{j=1}^{n})) \\
con_i := con_i'\,(var_{i1}, \ldots, var_{in_i}) \\
\\
\langle \Gamma' \vdash_{\mathrm{eq}} \forall(var_{i1}, \ldots, var_{in_i}).con_i \rightsquigarrow \Lambda(var_{i1}^{\langle\mathrm{eq}\rangle_{(i1)'}}, \ldots, var_{in_i}^{\langle\mathrm{eq}\rangle_{(in_i)'}}).exp_i^{\mathrm{eq}}\rangle_{i'} \\
\\
mod_i := [\overline{tycon_i}\triangleright var_i^{dt}{=}con_i', \\
\qquad \langle \mathrm{eq}{=}\Lambda(var_{i1}^{\langle\mathrm{eq}\rangle_{(i1)'}}, \ldots, var_{in_i}^{\langle\mathrm{eq}\rangle_{(in_i)'}}).exp_i^{\mathrm{eq}}\rangle_{i'} \\
\qquad \overline{id_{ij}}{=}\Lambda(var_{i1}, \ldots, var_{in_i}).\left\{\begin{array}{c}\mathsf{roll}^{con_i}\,(\mathsf{inj}_{\overline{id_{ij}}}^{con_i^{\mathrm{sum}}}\ \{\})\\ \text{or}\\ \lambda(var{:}con_{ij}){:}con_i.\mathsf{roll}^{con_i}\,(\mathsf{inj}_{\overline{id_{ij}}}^{con_i^{\mathrm{sum}}}\ var)\end{array}\right\}_{ij}, \quad (\text{for } j \in 1..m_i) \\
\qquad \mathrm{expose}{=}\Lambda(var_{i1}, \ldots, var_{in_i}).\lambda(var{:}con_i){:}con_i^{\mathrm{sum}}.\mathsf{unroll}\ var] : sig_i \\
\\
sig_i := [\overline{tycon_i}\triangleright var_i{:}\Omega^{n_i}\Rightarrow_i\Omega{=}var_i^{dt}, \\
\qquad \langle \mathrm{eq}{:}\forall(var_{i1}^{\langle\mathrm{eq}\rangle_{(i1)'}}, \ldots, var_{in_i}^{\langle\mathrm{eq}\rangle_{(in_i)'}}).var_i\times var_i \rightharpoonup \mathsf{Bool}\rangle_{i'}, \\
\qquad \overline{id_{ij}}{:}\forall(var_{i1}, \ldots, var_{in_i}).\left\{\begin{array}{c}con_i\\ \text{or}\\ con_{ij}\to con_i\end{array}\right\}, \quad (\text{for } j \in 1..m_i) \\
\qquad \mathrm{expose}{:}\forall(var_{i1}, \ldots, var_{in_i}).con_i\to con_i^{\mathrm{sum}}] \\
mod := [\overline{tycon_1}\triangleright var_1^{dt}{=}con_1', \cdots, \overline{tycon_p}\triangleright var_p^{dt}{=}con_p', \overline{tycon_1}^{*}{=}mod_1, \ldots, \overline{tycon_p}^{*}{=}mod_p] \\
sig := [\overline{tycon_1}\triangleright var_1^{dt}{:}\Omega^{n_1}\Rightarrow\Omega, \ldots, \overline{tycon_p}\triangleright var_p^{dt}{:}\Omega^{n_p}\Rightarrow\Omega, \overline{tycon_1}^{*}{:}sig_1, \ldots, \overline{tycon_p}^{*}{:}sig_p]
\end{array}
}{
\begin{array}{l}
\Gamma \vdash \texttt{(}tyvar_{11}\texttt{,} \cdots \texttt{,}tyvar_{1n_1}\texttt{)}\ tycon_1\texttt{=}\ id_{11}\left\{\begin{array}{c}\text{or}\\ \texttt{of}\ ty_{11}\end{array}\right\}_{11}\texttt{|}\cdots\texttt{|}\,id_{1m_1}\left\{\begin{array}{c}\text{or}\\ \texttt{of}\ ty_{1m_1}\end{array}\right\}_{1m_1} \\
\qquad \texttt{and}\ \cdots \texttt{and} \\
\qquad \texttt{(}tyvar_{p1}\texttt{,} \cdots \texttt{,}tyvar_{pn_p}\texttt{)}\ tycon_p\texttt{=}id_{p1}\left\{\begin{array}{c}\text{or}\\ \texttt{of}\ ty_{p1}\end{array}\right\}_{p1}\texttt{|}\cdots\texttt{|}\,id_{pm_p}\left\{\begin{array}{c}\text{or}\\ \texttt{of}\ ty_{pm_p}\end{array}\right\}_{pm_p} \rightsquigarrow \\
\qquad 1^{*}{=}(mod{:}sig) : 1^{*}{:}sig
\end{array}
}
\tag{240}
$$

6.7 Polymorphic Instantiation

$$\boxed{\mathit{decs} \vdash_{\mathrm{inst}}\leadsto [\mathit{sbnds}_{\mathrm{v}}] : [\mathit{sdecs}']}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{con} : \Omega \\ \langle \mathit{decs} \vdash_{\mathrm{eq}} \mathit{con} \leadsto \mathit{exp}_{\mathrm{v}} \rangle \\ \langle\langle \mathit{decs} \vdash_{\mathrm{inst}}\leadsto [\mathit{sbnds}_{\mathrm{v}}] : [\mathit{sdecs}] \rangle\rangle \end{array}}{\begin{array}{l} \mathit{decs} \vdash_{\mathrm{inst}}\leadsto [\mathit{lab}'{=}[\mathit{lab}\triangleright \mathit{var}{=}\mathit{con}\langle, \mathrm{eq}{=}\mathit{exp}_{\mathrm{v}}\rangle]\langle\langle, \mathit{sbnds}_{\mathrm{v}}\rangle\rangle] : \\ \qquad [\mathit{lab}'{:}[\mathit{lab}\triangleright \mathit{var}{:}\Omega{=}\mathit{con}\langle, \mathrm{eq}{:}\mathit{var}\times \mathit{var}\rightharpoonup \mathsf{Bool}\rangle]]\langle\langle, \mathit{sdecs}\rangle\rangle \end{array}} \quad (241)$$

Rule 241 Nondeterministically choose types and the corresponding equality functions so as to match a fully-transparent signature.

6.8 Equality Compilation

$$\boxed{\mathit{decs} \vdash_{\mathrm{eq}} \mathit{con} \leadsto \mathit{exp}}$$

$$\frac{}{\mathit{decs} \vdash_{\mathrm{eq}} \mathsf{Int} \leadsto \lambda(\mathit{var}_1{:}\mathsf{Int}, \mathit{var}_2{:}\mathsf{Int}{:}).\mathsf{Bool}(\mathit{var}_1 =_{\mathsf{Int}} \mathit{var}_2)} \quad (242)$$

Rule 242: We assume primitive equality operations already exist at type Int, with similar rules for the other base types.

$$\frac{\begin{array}{c} \mathit{con} = \{\mathit{lab}_1{:}\mathit{con}_1, \ldots, \mathit{lab}_n{:}\mathit{con}_n\} \\ \mathit{decs} \vdash_{\mathrm{eq}} \mathit{con}_1 \leadsto \mathit{exp}_1 \quad \cdots \quad \mathit{decs} \vdash_{\mathrm{eq}} \mathit{con}_n \leadsto \mathit{exp}_n \end{array}}{\begin{array}{l} \mathit{decs} \vdash_{\mathrm{eq}} \mathit{con} \leadsto \\ \quad \lambda(\mathit{var}_1{:}\mathit{con}, \mathit{var}_2{:}\mathit{con}){:}\mathsf{Bool}.\mathit{exp}_1\,(\pi_{\mathit{lab}_1}\,\mathit{var}_1, \pi_{\mathit{lab}_1}\,\mathit{var}_2)\ \mathsf{and} \\ \qquad\qquad\qquad \cdots\ \mathsf{and}\ \mathit{exp}_n\,(\pi_{\mathit{lab}_n}\,\mathit{var}_1, \pi_{\mathit{lab}_n}\,\mathit{var}_2) \end{array}} \quad (243)$$

Rule 243: Records are compared component-wise for equality.

$$\frac{\begin{array}{l} \qquad \forall i \in 1..n : \quad \mathit{con}_{\langle \mathit{lab}\rangle} := \Sigma_{\langle \mathit{lbl}\rangle}\,(\mathit{con}_1, \ldots, \mathit{con}_n) \\ \left\{ \begin{array}{l} \mathit{decs} \vdash_{\mathrm{eq}} \mathit{con}_i \leadsto \mathit{exp}_i \\ \quad \mathit{exp}'_i = \lambda \mathit{var}'{:}\mathit{con}_{\mathit{lab}_i}.\mathit{exp}_i\,(\mathsf{proj}^{\mathit{con}}_{\mathit{lab}_i}\,\mathit{var}', \mathsf{pproj}^{\mathit{con}}_{\mathit{lab}_i}\,(\mathit{var}_2,)) \end{array} \right. \end{array}}{\begin{array}{l} \mathit{decs} \vdash_{\mathrm{eq}} \mathit{con} \leadsto \\ \quad \lambda(\mathit{var}_1{:}\mathit{con}, \mathit{var}_2{:}\mathit{con}){:}\mathsf{Bool}.\mathsf{catch\ case}^{\mathit{con}}\ \mathit{var}_1\ \mathsf{of}\,\mathit{exp}'_1, \ldots, \mathit{exp}'_n\ \mathsf{end\,with\ false} \end{array}} \quad (244)$$

Rule 244: Values of a sum type are equal if they have the same tag and the tagged values are equal.

$$\frac{\begin{array}{c} k \in 1..n \\ \forall i \in 1..n : \quad \begin{cases} \mathit{con}_i = (\pi_i\,(\mu\,\mathit{con}'))\;\mathit{con}'' \\ \mathit{con}'_i := (\pi_i\,(\mathit{con}'\,(\mu\,\mathit{con}')))\;\mathit{con}'' \end{cases} \\ \mathit{decs}' = \mathit{decs}, \mathit{var}_1{:}[1{\rhd}\mathit{var}{:}\Omega{=}\mathit{con}_1, \mathrm{eq}{:}\mathit{var}\times\mathit{var}{\rightharpoonup}\mathsf{Bool}], \ldots, \\ \mathit{var}_n{:}[1{\rhd}\mathit{var}{:}\Omega{=}\mathit{con}_n, \mathrm{eq}{:}\mathit{var}\times\mathit{var}{\rightharpoonup}\mathsf{Bool}] \\ \forall i \in 1..n : \quad \begin{cases} \mathit{decs}' \vdash_{\mathsf{eq}} \mathsf{Unroll}\;\mathit{con}'_i \rightsquigarrow \mathit{exp}_i \\ \mathit{exp}_i^{\mathrm{eq}} = (\{\mathit{var}_1^{\mathrm{eq}}/\mathit{var}'_1.\mathrm{eq}\}\cdots\{\mathit{var}_n^{\mathrm{eq}}/\mathit{var}'_n.\mathrm{eq}\}\,\mathit{exp}_i) \\ \qquad (\mathsf{unroll}\,(\pi_{\bar{1}}\,\mathit{var}), \mathsf{unroll}\,(\pi_{\bar{2}}\,\mathit{var})) \end{cases} \end{array}}{\mathit{decs} \vdash_{\mathsf{eq}} \mathit{con}_k \rightsquigarrow \mathsf{fix}\,(\mathit{var}_i^{\mathrm{eq}}(\mathit{var}{:}\mathit{con}_i\times\mathit{con}_i){:}\mathsf{Bool}{\mapsto}\mathit{exp}_i^{\mathrm{eq}})_{i=1}^{n}\,\mathsf{end}\,\mathit{var}_j^{\mathrm{eq}}} \qquad (245)$$

Rule 245: A recursive type generates recursively-defined equality functions.

$$\frac{\mathit{decs} \vdash \mathit{path}.\mathrm{eq} : \mathit{path.lab}\times\mathit{path.lab}{\rightharpoonup}\mathsf{Bool}}{\mathit{decs} \vdash_{\mathsf{eq}} \mathit{path.lab} \rightsquigarrow \mathit{path}.\mathrm{eq}} \qquad (246)$$

Rule 246: If the constructor is simply a path—the name of an abstract type—we look for an equality function for that type in the same structure.

$$\frac{\begin{array}{c} \mathit{con} = \mathit{path.lab}\,(\mathit{con}_1, \ldots, \mathit{con}_n) \\ \mathit{decs} \vdash_{\overline{\mathsf{inst}}}{\rightsquigarrow}\; \mathit{mod}_p : \mathit{sig}_p \\ \mathit{decs} \vdash \mathit{path}.\mathrm{eq} : \mathit{sig}_p{\to}[\mathrm{it}{:}\mathit{con}\times\mathit{con}{\rightharpoonup}\mathsf{Bool}] \end{array}}{\mathit{decs} \vdash_{\mathsf{eq}} \mathit{con} \rightsquigarrow (\mathit{path}.\mathrm{eq}\,(\mathit{mod}_{\mathrm{v}})).\mathrm{it}} \qquad (247)$$

Rule 247: Same as Rule 246, except here we deal with the application of an abstract type to a tuple of types. The instantiation judgment may recursively invoke the equality compiler, in order to create equality functions for $\mathit{con}_1, \ldots, \mathit{con}_n$.

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega \\ \mathit{decs} \vdash_{\mathsf{eq}} \mathit{con}' \rightsquigarrow \mathit{exp} \end{array}}{\mathit{decs} \vdash_{\mathsf{eq}} \mathit{con} \rightsquigarrow \mathit{exp}} \qquad (248)$$

$$\boxed{\mathit{decs} \vdash_{\mathsf{eq}} \mathit{sig} \rightsquigarrow \mathit{mod}}$$

$$\frac{\begin{array}{c} \mathit{sig} = [1^*{:}[1{\rhd}\mathit{var}_1{:}\Omega], \ldots, n^*{:}[n{\rhd}\mathit{var}_n{:}\Omega]] \\ \mathit{sig}' := [1^*{:}[1{\rhd}\mathit{var}_1{:}\Omega\langle, \mathrm{eq}{:}\mathit{var}_1\times\mathit{var}_1{\rightharpoonup}\mathsf{Bool}\rangle_1], \ldots, n^*{:}[n{\rhd}\mathit{var}_n{:}\Omega\langle, \mathrm{eq}{:}\mathit{var}_n\times\mathit{var}_n{\rightharpoonup}\mathsf{Bool}\rangle_n]] \\ \mathit{decs}, \mathit{var}{:}\mathit{sig}' \vdash_{\mathsf{eq}} \mathit{con} \rightsquigarrow \mathit{exp} \end{array}}{\mathit{decs} \vdash_{\mathsf{eq}} \mathit{var}{:}\mathit{sig}{\to}[\mathrm{it}{:}\mathit{con}] \rightsquigarrow \lambda\mathit{var}{:}\mathit{sig}'.[\mathrm{it}{=}\mathit{exp}]} \qquad (249)$$

6.9 Pattern Compilation

Patterns

$$\boxed{\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \textsf{else}\ \mathit{exp}' \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}$$

The judgment should be read as "the bindings of the result of matching *exp* to the pattern *pat* are *sbnds*."

$$\frac{\begin{array}{c}\text{Rules 255Γ257Γand 259 do not apply.}\\ \Gamma \vdash \mathit{exp} : \mathit{con}\end{array}}{\Gamma \vdash \mathit{id} \Leftarrow \mathit{exp} : \mathit{con}\ \textsf{else}\ \mathit{exp}' \rightsquigarrow \overline{\mathit{id}}{=}\mathit{exp} : \overline{\mathit{id}}{:}\mathit{con}} \tag{250}$$

Rule 250: Pattern match against an variable (which is not a datatype or exception constructor).

$$\frac{\begin{array}{c}\mathit{lab}\ \text{fresh} \qquad \text{type}(\mathit{scon}) = \mathit{con}\\ \Gamma \vdash \mathit{exp} : \mathit{con}\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{scon} \Leftarrow \mathit{exp} : \mathit{con}\ \textsf{else}\ \mathit{exp}' \rightsquigarrow\\ \qquad \mathit{lab}{=}\textsf{if}\ \mathit{exp}{=}_{\mathit{con}}\mathit{scon}\ \textsf{then}\ \{\}\ \textsf{else}\ \textsf{raise}^{\textsf{Unit}}\ \mathit{exp}' : \mathit{lab}{:}\textsf{Unit}\end{array}} \tag{251}$$

Rule 251: Pattern match against a constant. We need primitive equality functions for constants which can appear in patterns.

$$\frac{\mathit{lab}\ \text{fresh} \qquad \Gamma \vdash \mathit{exp} : \mathit{con}}{\Gamma \vdash \_ \Leftarrow \mathit{exp} : \mathit{con}\ \textsf{else}\ \mathit{exp}' \rightsquigarrow \mathit{lab}{=}\mathit{exp} : \mathit{lab}{:}\mathit{con}} \tag{252}$$

Rule 252: Pattern match against a wildcard.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con}' : \Omega \qquad \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \Omega\\ \Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \textsf{else}\ \mathit{exp}' \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}\end{array}}{\Gamma \vdash \mathit{pat}\ :\ \mathit{ty} \Leftarrow \mathit{exp} : \mathit{con}\ \textsf{else}\ \mathit{exp}' \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}} \tag{253}$$

Rule 253: Pattern match against an explicitly-typed pattern.

$$\frac{\begin{array}{c}\Gamma \vdash_{\textsf{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path}.\mathit{lab}_i : \mathit{con}_i{\rightarrow}\mathit{con}\\ \Gamma \vdash \mathit{path}.\text{expose} : \mathit{con}{\rightarrow}\Sigma(\mathit{lab}_1{\mapsto}\mathit{con}_1, \ldots, \mathit{lab}_n{\mapsto}\mathit{con}_n)\\ \Gamma \vdash \mathit{pat} \Leftarrow \textsf{pproj}_{\mathit{lab}_i}^{\Sigma(\mathit{lab}_1\mapsto \mathit{con}_1,\ldots,\mathit{lab}_n\mapsto \mathit{con}_n)}\ ((\mathit{path}.\text{expose}\ \mathit{exp}), \mathit{exp}') : \mathit{con}_i\ \textsf{else}\ \mathit{exp}' \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}\end{array}}{\Gamma \vdash \mathit{longid}\ \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \textsf{else}\ \mathit{exp}' \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}} \tag{254}$$

Rule 254: Pattern match against a monomorphic datatype constructor which carries a value.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path.lab_i : con \qquad lab \text{ fresh} \\ \Gamma \vdash exp : con \\ \Gamma \vdash path.\mathrm{expose} : con \to \Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)\end{array}}{\begin{array}{l}\Gamma \vdash longid \Leftarrow exp : con \;\mathsf{else}\; exp' \rightsquigarrow \\ \quad lab{=}\mathsf{pproj}_{lab_i}^{\Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)}\,((path.\mathrm{expose}\; exp), exp') : lab{:}\mathsf{Unit}\end{array}} \qquad (255)$$

Rule 255: Pattern match against a constant, monomorphic datatype constructor.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path.lab_i : sig \\ \Gamma \vdash path.\mathrm{expose} : sig_p \to [\mathrm{it}{:}con \to \Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)] \\ \Gamma \vdash_{\mathrm{inst}} \rightsquigarrow mod_p : sig_p \\ \Gamma \vdash pat \Leftarrow \mathsf{pproj}_{lab_i}^{\Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)}\,(((path.\mathrm{expose}\; mod_p).\mathrm{it}\; exp), exp') : con_i \;\mathsf{else}\; exp' \rightsquigarrow \\ \quad sbnds : sdecs\end{array}}{\Gamma \vdash longid\; pat \Leftarrow exp : con \;\mathsf{else}\; exp' \rightsquigarrow sbnds : sdecs} \qquad (256)$$

Rule 256: Pattern match against a polymorphic datatype constructor carrying a value.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path.lab_i : sig \\ \Gamma \vdash exp : con \\ \Gamma \vdash path.\mathrm{expose} : sig_p \to [\mathrm{it}{:}con \to \Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)] \\ \Gamma \vdash_{\mathrm{inst}} \rightsquigarrow mod_p : sig_p \qquad lab \text{ fresh}\end{array}}{\begin{array}{l}\Gamma \vdash longid \Leftarrow exp : con \;\mathsf{else}\; exp' \rightsquigarrow \\ \quad lab{=}\mathsf{pproj}_{lab_i}^{\Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)}\,((path.\mathrm{expose}\; mod_p).\mathrm{it}\; exp, exp') : lab{:}\mathsf{Unit}\end{array}} \qquad (257)$$

Rule 257: Pattern match against a constant polymorphic constructor.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path.lab : con \to \mathsf{Tagged} \\ \Gamma \vdash path.\mathrm{tag} : con\; \mathsf{Tag} \\ \Gamma \vdash pat \Leftarrow (\mathsf{iftagof}\; exp\; \mathsf{is}\; path.\mathrm{tag}\; \mathsf{then}\; \lambda var{:}con.var\; \mathsf{else}\; \mathsf{raise}^{con}\; exp') \;\mathsf{else}\; exp' \rightsquigarrow \\ \quad sbnds : sdecs\end{array}}{\Gamma \vdash longid\; pat \Leftarrow exp : con \;\mathsf{else}\; exp' \rightsquigarrow sbnds : sdecs} \qquad (258)$$

Rule 258: Pattern match against value-carrying exception constructor.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path.lab : \mathsf{Tagged} \qquad lab \text{ fresh} \\ \Gamma \vdash exp : con \\ \Gamma \vdash path.\mathrm{tag} : \mathsf{Unit}\; \mathsf{Tag}\end{array}}{\begin{array}{l}\Gamma \vdash longid \Leftarrow exp : con \;\mathsf{else}\; exp' \rightsquigarrow \\ \quad lab{=}\mathsf{iftagof}\; exp\; \mathsf{is}\; path.\mathrm{tag}\; \mathsf{then}\; \lambda var{:}\mathsf{Unit}.\{\}\; \mathsf{else}\; \mathsf{raise}^{\mathsf{Unit}}\; exp' : lab{:}\mathsf{Unit}\end{array}} \qquad (259)$$

Rule 259: Pattern match against constant exception constructor.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{con} \equiv \{\mathit{lab}'_1{:}\mathit{con}'_1, \cdots, \mathit{lab}'_k{:}\mathit{con}'_k\} : \Omega \\ \{\overline{\mathit{lab}_1}, \cdots, \overline{\mathit{lab}_n}\} \subseteq \{\mathit{lab}'_1, \cdots, \mathit{lab}'_n\} \\ \forall i \in 1..n : \quad \Gamma, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con} \vdash \mathit{pat}_i \Leftarrow \pi_{\overline{\mathit{lab}_i}}\, \mathit{exp} : \mathit{con}_i \text{ else } \mathit{exp}' \leadsto \mathit{sbnds}_i : \mathit{sdecs}_i\end{array}}{\begin{array}{c}\Gamma \vdash \{\mathit{lab}_1 = \mathit{pat}_1, \ldots, \mathit{lab}_n = \mathit{pat}_n\langle, \ldots\rangle\} \Leftarrow \mathit{exp} : \mathit{con} \text{ else } \mathit{exp}' \leadsto \\ \mathit{sbnds}_1, \ldots, \mathit{sbnds}_n : \mathit{sdecs}_1, \ldots, \mathit{sdecs}_n\end{array}} \qquad (260)$$

Rule 260: Pattern match against a record of patterns. Because we disallow repeated variables in patterns, the syntactic concatenation of structure here is well-formed.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{pat}_1 \Leftarrow \mathit{exp} : \mathit{con} \text{ else } \mathit{exp}' \leadsto \mathit{sbnds}_1 : \mathit{sdecs}_1 \\ \Gamma \vdash \mathit{pat}_2 \Leftarrow \mathit{exp} : \mathit{con} \text{ else } \mathit{exp}' \leadsto \mathit{sbnds}_2 : \mathit{sdecs}_2\end{array}}{\Gamma \vdash \mathit{pat}_1 \text{ as } \mathit{pat}_2 \Leftarrow \mathit{exp} : \mathit{con} \text{ else } \mathit{exp}' \leadsto \mathit{sbnds}_1, \mathit{sbnds}_2 : \mathit{sdecs}_1, \mathit{sdecs}_2} \qquad (261)$$

Rule 261: Pattern match against two patterns simultaneously.

$$\frac{\Gamma \vdash \mathit{pat} \Leftarrow \text{get}\, \mathit{exp} : \mathit{con} \text{ else } \mathit{exp}' \leadsto \mathit{sbnds} : \mathit{sdecs}}{\Gamma \vdash \mathtt{ref}\ \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\, \text{Ref else } \mathit{exp}' \leadsto \mathit{sbnds} : \mathit{sdecs}} \qquad (262)$$

Rule 262: Pattern match involving implicit dereferencing of a `ref` cell.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}' \text{ else } \mathit{exp}' \leadsto \mathit{sbnds} : \mathit{sdecs} \\ \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \Omega\end{array}}{\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con} \text{ else } \mathit{exp}' \leadsto \mathit{sbnds} : \mathit{sdecs}} \qquad (263)$$

6.10 Coercion Compilation

$$\boxed{\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{sdec} \leadsto \mathit{sbnd} : \mathit{sdec}'}$$

$$\frac{\mathit{decs}; \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{sig}} \mathit{lab} \leadsto \mathit{labs} : \mathit{con}'}{\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con} \leadsto \mathit{lab}{\triangleright}\mathit{var}{=}\mathit{var}_0.\mathit{labs} : \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}} \tag{264}$$

Rule 264: Coercion of a monomorphic value specification to a monomorphic value specification.

$$\frac{\mathit{decs}; \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{sig}} \mathit{lab} \leadsto \mathit{labs} : \mathit{con}{\rightarrow}\mathit{con}'}{\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}{\rightharpoonup}\mathit{con}' \leadsto \mathit{lab}{\triangleright}\mathit{var}{=}\partial\, \mathit{var}_0.\mathit{labs} : \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}{\rightharpoonup}\mathit{con}'} \tag{265}$$

$$\frac{\begin{array}{c}\mathit{decs}, \mathit{var}_p'{:}\mathit{sig}_p'; \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{sig}} \mathit{lab} \leadsto \mathit{labs} : \mathit{sig}_p{\rightarrow}[\mathrm{it}{:}\mathit{con}] \\ \mathit{decs}, \mathit{var}_p'{:}\mathit{sig}_p', \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{inst}\leadsto} \mathit{mod}_p : \mathit{sig}_p\end{array}}{\begin{array}{l}\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{lab}{\triangleright}\mathit{var}{:}(\mathit{var}_p'{:}\mathit{sig}_p'){\rightarrow}[\mathrm{it}{:}\mathit{con}] \leadsto \\ \quad \mathit{lab}{\triangleright}\mathit{var}{=}\lambda \mathit{var}_p'{:}\mathit{sig}_p'.(\mathit{var}_0.\mathit{labs}\ \mathit{mod}_p) : \\ \quad \mathit{lab}{\triangleright}\mathit{var}{:}(\mathit{var}_p'{:}\mathit{sig}_p'){\rightarrow}[\mathrm{it}{:}\mathit{con}]\end{array}} \tag{266}$$

Rule 266: Coercion of a polymorphic value specification to a polymorphic value specification; this may involve implicit polymorphic instantiation. (The rule also handles matching polymorphic datatype constructors to `val` specifications.) Note that sig_p and sig_p' need not have the same labels, so this rule also handles alpha-conversion of EL type variables.

$$\frac{\begin{array}{c}\mathit{decs}, \mathit{var}_p'{:}\mathit{sig}_p'; \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{sig}} \mathit{lab} \leadsto \mathit{labs} : \mathit{sig}_p{\rightarrow}[\mathrm{it}{:}\mathit{con}{\rightarrow}\mathit{con}'] \\ \mathit{decs}, \mathit{var}_p'{:}\mathit{sig}_p', \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{inst}\leadsto} \mathit{mod}_p : \mathit{sig}_p\end{array}}{\begin{array}{l}\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{lab}{\triangleright}\mathit{var}{:}(\mathit{var}_p'{:}\mathit{sig}_p'){\rightarrow}[\mathrm{it}{:}\mathit{con}{\rightharpoonup}\mathit{con}'] \leadsto \\ \quad \mathit{lab}{\triangleright}\mathit{var}{=}\lambda \mathit{var}_p'{:}\mathit{sig}_p'.[\mathrm{it}{=}\partial\left((\mathit{var}_0.\mathit{labs}\ \mathit{mod}_p).\mathrm{it}\right)] : \\ \quad \mathit{lab}{\triangleright}\mathit{var}{:}(\mathit{var}_p'{:}\mathit{sig}_p'){\rightarrow}[\mathrm{it}{:}\mathit{con}{\rightharpoonup}\mathit{con}']\end{array}} \tag{267}$$

$$\frac{\begin{array}{c}\mathit{decs}; \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{sig}} \mathit{lab} \leadsto \mathit{labs} : \mathit{var}_p{:}\mathit{sig}{\rightarrow}[\mathrm{it}{:}\mathit{con}] \\ \mathit{decs} \vdash_{\mathrm{inst}\leadsto} \mathit{mod} : \mathit{sig}\end{array}}{\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con} \leadsto \mathit{lab}{\triangleright}\mathit{var}{=}(\mathit{var}_0.\mathit{labs}\ \mathit{mod}).\mathrm{it} : \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}} \tag{268}$$

Rule 268: Coercion of a polymorphic value (or datatype constructor) to match a monomorphic value specification.

$$\frac{\begin{array}{c}\mathit{decs}; \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{sig}} \mathit{lab} \leadsto \mathit{labs} : \mathit{var}_p{:}\mathit{sig}{\rightarrow}[\mathrm{it}{:}\mathit{con}{\rightarrow}\mathit{con}'] \\ \mathit{decs} \vdash_{\mathrm{inst}\leadsto} \mathit{mod} : \mathit{sig}\end{array}}{\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}{\rightharpoonup}\mathit{con}' \leadsto \mathit{lab}{\triangleright}\mathit{var}{=}\partial\left((\mathit{var}_0.\mathit{labs}\ \mathit{mod}).\mathrm{it}\right) : \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{con}{\rightharpoonup}\mathit{con}'} \tag{269}$$

$$\frac{\begin{array}{c}\mathit{decs}; \mathit{var}_0{:}\mathit{sig}_0 \vdash_{\mathrm{sig}} \mathit{lab} \leadsto \mathit{labs} : \mathit{knd} \\ \langle \mathit{decs}, \mathit{var}_0{:}\mathit{sig}_0 \vdash \mathit{var}_0.\mathit{labs} \equiv \mathit{con} : \mathit{knd}\rangle\end{array}}{\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}\langle{=}\mathit{con}\rangle \leadsto \mathit{lab}{\triangleright}\mathit{var}{=}\mathit{var}_0.\mathit{labs} : \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{var}_0.\mathit{labs}} \tag{270}$$

Rule 270: Coercion of a type component to a `type` specification.

$$\frac{\begin{array}{c} decs; var_0{:}sig_0 \vdash_{\mathrm{sig}} lab \leadsto labs : \Omega \\ \langle decs, var_0{:}sig_0 \vdash var_0.labs \equiv con : \Omega\rangle \\ decs, var_0{:}sig_0 \vdash_{\mathrm{eq}} var_0.labs \leadsto exp \end{array}}{\begin{array}{l} decs; path{:}sig_0 \vdash_{\mathrm{sub}} lab^*{:}[lab{\triangleright}var{:}\Omega\langle{=}con\rangle, \mathrm{eq}{:}var\times var \rightharpoonup \mathsf{Bool}] \leadsto \\ \quad lab'^*{:}[lab{\triangleright}var{=}var_0.labs, \mathrm{eq}{:}exp] : \\ \quad lab'^*{:}[lab{\triangleright}var{:}\Omega{=}var_0.labs, \mathrm{eq}{:}var\times var \rightharpoonup \mathsf{Bool}] \end{array}} \tag{271}$$

Coercion of a type declaration corresponding to an `eqtype` signature specification involving no type variables. We invoke the equality compiler to create an equality function for the type being copied.

$$\frac{\begin{array}{c} decs; var_0{:}sig_0 \vdash_{\mathrm{sig}} lab \leadsto labs : \Omega^n{\Rightarrow}\Omega \\ con' = var(var_p.lab'_1.lab_1, \cdots, var_p.lab'_n.lab_n) \\ decs, var_0{:}sig_0, var_p{:}sig_p \vdash_{\mathrm{eq}} con' \leadsto exp \end{array}}{\begin{array}{l} decs; path{:}sig_0 \vdash_{\mathrm{sub}} lab'^*{:}[lab{\triangleright}var{:}\Omega^n{\Rightarrow}\Omega, \mathrm{eq}{:}var_p{:}sig_p{\to}[\mathrm{it}{:}con'\times con' \rightharpoonup \mathsf{Bool}]] \leadsto \\ \quad lab^*{:}[lab{\triangleright}var{=}var_0.labs, \mathrm{eq}{:}\lambda var_p{:}sig_p.[\mathrm{it}{=}exp]] : \\ \quad lab^*{:}[lab{\triangleright}var{:}\Omega^n{\Rightarrow}\Omega\langle{=}var_0.labs\rangle, \mathrm{eq}{:}var_p{:}sig_p{\to}[\mathrm{it}{:}con'\times con' \rightharpoonup \mathsf{Bool}]] \end{array}} \tag{272}$$

Coercion of a type declaration correspond to a `eqtype` signature specification involving type variables (i.e., `eqtype ('a,'b) foo`).

$$\frac{\begin{array}{c} decs; var_0{:}sig_0 \vdash_{\mathrm{sig}} lab \leadsto labs : sig' \\ decs, var_0{:}sig_0 \vdash_{\mathrm{sub}} path.labs : sig' \preceq sig \leadsto mod_c : sig_c \end{array}}{decs; path{:}sig_0 \vdash_{\mathrm{sub}} lab{\triangleright}var{:}sig \leadsto lab{\triangleright}var{=}mod_c : lab{\triangleright}var{:}sig_c} \tag{273}$$

Rule 273: Coercion of a module component.

$$\boxed{decs \vdash_{\mathrm{sub}} path : sig_0 \preceq sig \leadsto mod : sig'}$$

$$\frac{\begin{array}{c} decs; path{:}sig_0 \vdash_{\mathrm{sub}} lab_1{\triangleright}dec_1 \leadsto sbnd_1 : sdec_1 \\ decs, dec_1; path{:}sig_0 \vdash_{\mathrm{sub}} lab_2{\triangleright}dec_2 \leadsto sbnd_2 : sdec_2 \\ \vdots \\ decs, dec_1, \ldots, dec_{n-1}; path{:}sig_0 \vdash_{\mathrm{sub}} lab_n{\triangleright}dec_n \leadsto sbnd_n : sdec_n \end{array}}{\begin{array}{c} decs \vdash_{\mathrm{sub}} path : sig_0 \preceq [lab_1{\triangleright}dec_1, \ldots, lab_n{\triangleright}dec_n] \leadsto \\ [sbnd_1, \ldots, sbnd_n] : [sdec_1, \ldots, sdec_n] \end{array}} \tag{274}$$

$$\frac{\begin{array}{c} decs \vdash_{\mathrm{sub}} var_2 : sig_2 \preceq sig_1 \leadsto mod_3 : sig_3 \\ decs, var_2{:}sig_2 \vdash_{\mathrm{sub}} var'_1 : sig'_1 \preceq sig'_2 \leadsto mod_4 : sig_4 \\ decs, var_2{:}sig_2 \vdash mod_4 : sig_4 \end{array}}{\begin{array}{l} decs \vdash_{\mathrm{sub}} path : (var_1{:}sig_1 \rightharpoonup sig'_1) \preceq (var_2{:}sig_2 \rightharpoonup sig'_2) \leadsto \\ \quad \lambda var_2{:}sig_2.\mathsf{let}\ var'_1{=}path\ mod_3\ \mathsf{in}\ mod_4\ \mathsf{end} : \\ \quad var_2{:}sig_2 \rightharpoonup sig_4 \end{array}} \tag{275}$$

Rule 275: Coercions for functor subtyping. We also need the same rule for subtyping of total functors (not shown here).

6.11 Signature Patching

where type

$$\boxed{\mathit{sig} \vdash_{\mathrm{wt}} \mathit{labs} := \mathit{con} : \mathit{knd} \leadsto \mathit{sig}' : \mathsf{Sig}}$$

This judgment should be read "By adding to the signature *sig* the fact that the abstract type component selected by *labs* is equal to *con* : *knd*, we get the signature *sig*′."

$$\frac{\begin{array}{c}\mathrm{FV}(\mathit{con}) \cap \mathrm{BV}(\mathit{sdecs}) = \emptyset \\ \mathit{sig} = [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}']\end{array}}{\mathit{sig} \vdash_{\mathrm{wt}} \mathit{lab} := \mathit{con} : \mathit{knd} \leadsto [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs}'] : \mathsf{Sig}} \tag{276}$$

$$\frac{\begin{array}{c}\mathrm{FV}(\mathit{con}) \cap \mathrm{BV}(\mathit{sdecs}) = \emptyset \\ \mathit{sig} = [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}'] \\ \mathit{sig}' \vdash_{\mathrm{wt}} \mathit{labs} := \mathit{con} : \mathit{knd} \leadsto \mathit{sig}'' : \mathsf{Sig}\end{array}}{\mathit{sig} \vdash_{\mathrm{wt}} \mathit{lab}.\mathit{labs} := \mathit{con} : \mathit{knd} \leadsto [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}'', \mathit{sdecs}'] : \mathsf{Sig}} \tag{277}$$

sharing

$$\boxed{\mathit{sig} \vdash_{\mathrm{sh}} \mathit{labs}_1 := \mathit{labs}_2 : \mathit{knd} \leadsto \mathit{sig}' : \mathsf{Sig}}$$

This judgment should be read "By adding to the signature *sig* the fact that the (abstract) type components of kind *knd* selected by labs_1 and labs_2 are equal, we get the signature *sig*′."

$$\frac{\mathit{sig} = [\mathit{sdecs}, \mathit{lab}'{\triangleright}\mathit{var}'{:}\mathit{knd}, \mathit{sdecs}', \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}'']}{\begin{array}{l}\mathit{sig} \vdash_{\mathrm{sh}} \mathit{lab} := \mathit{lab}' : \mathit{knd} \leadsto \\ \quad [\mathit{sdecs}, \mathit{lab}'{\triangleright}\mathit{var}'{:}\mathit{knd}, \mathit{sdecs}', \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{var}', \mathit{sdecs}'] : \mathsf{Sig}\end{array}} \tag{278}$$

$$\frac{\begin{array}{c}\mathit{sig} = [\mathit{sdecs}, \mathit{lab}'{\triangleright}\mathit{var}'{:}\mathit{knd}, \mathit{sdecs}', \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}, \mathit{sdecs}''] \\ \mathit{sig} \vdash_{\mathrm{wt}} \mathit{labs} := \mathit{var}' : \mathit{knd} \leadsto \mathit{sig}' : \mathsf{Sig}\end{array}}{\begin{array}{l}\mathit{sig} \vdash_{\mathrm{sh}} \mathit{lab}.\mathit{labs} := \mathit{lab}' : \mathit{knd} \leadsto \\ \quad [\mathit{sdecs}, \mathit{lab}'{\triangleright}\mathit{var}'{:}\mathit{knd}, \mathit{sdecs}', \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}'] : \mathsf{Sig}\end{array}} \tag{279}$$

$$\frac{\begin{array}{c}\mathit{sig} = [\mathit{sdecs}, \mathit{lab}'{\triangleright}\mathit{var}'{:}\mathit{sig}', \mathit{sdecs}', \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}'', \mathit{sdecs}''] \\ \mathit{sig}'' \vdash_{\mathrm{wt}} \mathit{labs} := \mathit{var}'.\mathit{labs}' : \mathit{knd} \leadsto \mathit{sig}''' : \mathsf{Sig}\end{array}}{\begin{array}{l}\mathit{sig} \vdash_{\mathrm{sh}} \mathit{lab}.\mathit{labs} := \mathit{lab}'.\mathit{labs}' : \mathit{knd} \leadsto \\ \quad [\mathit{sdecs}, \mathit{lab}'{\triangleright}\mathit{var}'{:}\mathit{sig}', \mathit{sdecs}', \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}''', \mathit{sdecs}''] : \mathsf{Sig}\end{array}} \tag{280}$$

$$\frac{\begin{array}{c}\mathit{sig} = [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}'] \\ \mathit{sig}' \vdash_{\mathrm{sh}} \mathit{labs} := \mathit{labs}' : \mathit{knd} \leadsto \mathit{sig}'' : \mathsf{Sig}\end{array}}{\mathit{sig} \vdash_{\mathrm{sh}} \mathit{lab}.\mathit{labs} := \mathit{lab}.\mathit{labs}' : \mathit{knd} \leadsto [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}'', \mathit{sdecs}'] : \mathsf{Sig}} \tag{281}$$

6.12 Lookup Rules

The lookup rules specify the order in which translation contexts and IL signatures are searched.

- To prevent an explosion of rulesΓthe metavariable *class* is used to denote a typeΓ signatureΓor kind as appropriate.
- Any IL structure label starred with an asterisk (e.g.Γlab^*) is treated specially by the lookupΓwhich checks inside the structure. That isΓwhen looking for an identifier in a contextΓwe also look *inside* starred structure declarations. If the identifier is found inside such a structureΓthe full path to the identifier through the open structure is returned.
- Any type or signature returned by a lookup will be valid with respect to the ambient context.

6.12.1 Context Lookup

$$\boxed{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path : class}$$

Main rules for looking up identifiers in a translation context.

$$\frac{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path \qquad \Gamma \vdash path : con}{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path : con} \tag{282}$$

$$\frac{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path \qquad \Gamma \vdash path : knd}{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path : knd} \tag{283}$$

$$\frac{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path \qquad \Gamma \vdash path : sig}{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path : sig} \tag{284}$$

$$\boxed{\Gamma \vdash_{\mathsf{ctx}} labs \leadsto path}$$

"Utility" rules used to look up identifiers in a translation context.

$$\frac{lab = lab'}{\Gamma, lab \triangleright var{:}con \vdash_{\mathsf{ctx}} lab' \leadsto var} \tag{285}$$

$$\frac{lab \neq lab' \qquad \Gamma \vdash_{\mathsf{ctx}} lab' \leadsto path}{\Gamma, lab \triangleright var{:}con \vdash_{\mathsf{ctx}} lab' \leadsto path} \tag{286}$$

$$\frac{lab = lab'}{\Gamma, lab \triangleright var{:}knd\langle{=}con\rangle \vdash_{\mathsf{ctx}} lab' \leadsto var} \tag{287}$$

$$\frac{lab \neq lab' \qquad \Gamma \vdash_{\mathsf{ctx}} lab' \leadsto path}{\Gamma, lab \triangleright var{:}knd\langle{=}con\rangle \vdash_{\mathsf{ctx}} lab' \leadsto path} \tag{288}$$

$$\frac{\mathit{lab} = \mathit{lab}'}{\Gamma, \mathit{lab} \rhd \mathit{var}{:}\mathit{sig} \vdash_{\mathsf{ctx}} \mathit{lab}' \leadsto \mathit{var}} \tag{289}$$

$$\frac{\mathit{lab} \neq \mathit{lab}' \qquad \Gamma \vdash_{\mathsf{ctx}} \mathit{lab}' \leadsto \mathit{path}}{\Gamma, \mathit{lab} \rhd \mathit{var}{:}\mathit{sig} \vdash_{\mathsf{ctx}} \mathit{lab}' \leadsto \mathit{path}} \tag{290}$$

$$\frac{\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab}' \leadsto \mathit{path}}{\Gamma, \mathit{lab}^* \rhd \mathit{var}{:}\mathit{sig} \vdash_{\mathsf{ctx}} \mathit{lab}' \leadsto \mathit{var}.\mathit{path}} \tag{291}$$

$$\frac{\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab}' \not\leadsto \qquad \Gamma \vdash_{\mathsf{ctx}} \mathit{lab}' \leadsto \mathit{path}}{\Gamma, \mathit{lab}^* \rhd \mathit{var}{:}\mathit{sig} \vdash_{\mathsf{ctx}} \mathit{lab}' \leadsto \mathit{path}} \tag{292}$$

$$\frac{\Gamma \vdash_{\mathsf{ctx}} \mathit{lab} \leadsto \mathit{path} : \mathit{sig} \qquad \Gamma; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{labs} \leadsto \mathit{labs}' : \mathit{class}}{\Gamma \vdash_{\mathsf{ctx}} \mathit{lab}.\mathit{labs} \leadsto \mathit{path}.\mathit{labs}'} \tag{293}$$

6.12.2 Signature Lookup

$$\boxed{\mathit{decs}; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{labs} \leadsto \mathit{labs}' : \mathit{class}}$$

Main rules for looking up identifiers in a signature.

$$\frac{\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig} \qquad \mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}' \qquad \mathit{decs} \vdash \mathit{path}.\mathit{labs}' : \mathit{con}}{\mathit{decs}; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}' : \mathit{con}} \tag{294}$$

$$\frac{\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig} \qquad \mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}' \qquad \mathit{decs} \vdash \mathit{path}.\mathit{labs}' : \mathit{sig}'}{\mathit{decs}; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}' : \mathit{sig}'} \tag{295}$$

$$\frac{\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig} \qquad \mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}' \qquad \mathit{decs} \vdash \mathit{path}.\mathit{labs}' : \mathit{knd}}{\mathit{decs}; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}' : \mathit{knd}} \tag{296}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{sig} : \mathsf{Sig} \\ \mathit{decs}; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}' : \mathit{sig}' \\ \mathit{decs}; \mathit{path}.\mathit{labs}'{:}\mathit{sig}' \vdash_{\mathsf{sig}} \mathit{labs} \leadsto \mathit{labs}'' : \mathit{class} \end{array}}{\mathit{decs}; \mathit{path}{:}\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab}.\mathit{labs} \leadsto \mathit{labs}'.\mathit{labs}'' : \mathit{class}} \tag{297}$$

$$\boxed{\mathit{sig} \vdash_{\mathsf{sig}} \mathit{lab} \leadsto \mathit{labs}'}$$

"Utility" rules used to look up identifiers in a signature.

$$\frac{\mathit{lab} = \mathit{lab}'}{[\mathit{sdecs}, \mathit{lab} \rhd \mathit{var}{:}\mathit{con}] \vdash_{\mathsf{sig}} \mathit{lab}' \leadsto \mathit{lab}} \tag{298}$$

$$\frac{\mathit{lab} \neq \mathit{lab}' \qquad [\mathit{sdecs}] \vdash_{\mathsf{sig}} \mathit{lab}' \leadsto \mathit{labs}}{[\mathit{sdecs}, \mathit{lab} \rhd \mathit{var}{:}\mathit{con}] \vdash_{\mathsf{sig}} \mathit{lab}' \leadsto \mathit{labs}} \tag{299}$$

$$\frac{lab = lab'}{[sdecs, lab \triangleright var{:}knd\langle{=}con\rangle] \vdash_{\mathsf{sig}} lab' \rightsquigarrow lab} \tag{300}$$

$$\frac{lab \neq lab' \qquad [sdecs] \vdash_{\mathsf{sig}} lab' \rightsquigarrow labs}{[sdecs, lab \triangleright var{:}knd\langle{=}con\rangle] \vdash_{\mathsf{sig}} lab' \rightsquigarrow labs} \tag{301}$$

$$\frac{lab = lab'}{[sdecs, lab \triangleright var{:}sig] \vdash_{\mathsf{sig}} lab' \rightsquigarrow lab} \tag{302}$$

$$\frac{lab \neq lab' \qquad [sdecs] \vdash_{\mathsf{sig}} lab' \rightsquigarrow labs}{[sdecs, lab \triangleright var{:}sig] \vdash_{\mathsf{sig}} lab' \rightsquigarrow labs} \tag{303}$$

$$\frac{sig \vdash_{\mathsf{sig}} lab' \rightsquigarrow labs}{[sdecs, lab^{*} \triangleright var{:}sig] \vdash_{\mathsf{sig}} lab' \rightsquigarrow lab^{*}.labs} \tag{304}$$

$$\frac{sig \vdash_{\mathsf{sig}} lab' \not\rightsquigarrow \qquad [sdecs] \vdash_{\mathsf{sig}} lab' \rightsquigarrow labs}{[sdecs, lab^{*} \triangleright var{:}sig] \vdash_{\mathsf{sig}} lab' \rightsquigarrow labs} \tag{305}$$

6.13 Overloading

For each overloaded identifier we wish to add to the ELΓwe add one translation rule for each overloaded type. For exampleΓwe may want to overload the EL identifier + for addition at types `int` and `real`. In this caseΓwe would add two rules along the lines of:

$$\frac{}{\Gamma \vdash + \rightsquigarrow +_{\mathsf{Int}} : \mathsf{Int} \times \mathsf{Int} \rightharpoonup \mathsf{Int}}$$

and

$$\frac{}{\Gamma \vdash + \rightsquigarrow +_{\mathsf{Float}} : \mathsf{Float} \times \mathsf{Float} \rightharpoonup \mathsf{Float}},$$

where $+_{\mathsf{Int}}$ and $+_{\mathsf{Float}}$ are the two IL primitive addition functions.

7 Properties

The proofs presented here are only sketches; the Standard ML language is far too large to check every case by hand in a reasonable amount of time. Future work may involve attempting to check these proofs with an automated system.

7.1 Properties of the Internal Language

7.1.1 Static Semantics

Lemma 1 (Well-formedness)
The following propositions hold:

1. *If* $decs \vdash dec$ *ok then* $\vdash decs$ *ok.*

2. *If* $decs \vdash bnd : dec$ *then* $decs \vdash dec$ *ok.*

3. *If* $decs \vdash knd : \mathsf{Kind}$ *then* $\vdash decs$ *ok.*

4. *If* $decs \vdash con : knd$ *then* $decs \vdash knd : \mathsf{Kind}$.

5. *If* $decs \vdash con \equiv con' : knd$ *then* $decs \vdash con : knd$ *and* $decs \vdash con' : knd$.

6. *If* $decs \vdash exp : con$ *then* $decs \vdash con : \Omega$.

7. *If* $decs \vdash sdecs$ *ok then* $\vdash decs$ *ok.*

8. *If* $decs \vdash sig : \mathsf{Sig}$ *then* $\vdash decs$ *ok.*

9. *If* $decs \vdash sdecs \leq sdecs'$ *then* $decs \vdash sdecs$ *ok and* $decs \vdash sdecs'$ *ok.*

10. *If* $decs \vdash sig \leq sig' : \mathsf{Sig}$ *then* $decs \vdash sig : \mathsf{Sig}$ *and* $decs \vdash sig' : \mathsf{Sig}$.

11. *If* $decs \vdash sdecs \equiv sdecs'$ *then* $decs \vdash sdecs$ *ok and* $decs \vdash sdecs'$ *ok.*

12. *If* $decs \vdash sig \equiv sig' : \mathsf{Sig}$ *then* $decs \vdash sig : \mathsf{Sig}$ *and* $decs \vdash sig' : \mathsf{Sig}$.

13. *If* $decs \vdash sbnds : sdecs$ *then* $decs \vdash sdecs$ *ok.*

14. *If* $decs \vdash mod : sig$ *then* $decs \vdash sig : \mathsf{Sig}$.

15. *If* $decs \vdash exp \downarrow con$ *then* $decs \vdash exp : con$. *If in addition* $decs \vdash exp : con'$, *then* $decs \vdash exp \downarrow con'$.

16. *If* $decs \vdash mod \downarrow sig$ *then* $decs \vdash mod : sig$. *If in addition* $decs \vdash mod : sig'$, *then* $decs \vdash mod \downarrow sig'$.

The following lemma states that internal language judgments are preserved under substitution of values for free variables in a typing judgmentΓwhere a value is defined syntactically in Figure 4 to be a phrase in evaluated form.

Proposition 2 (Substitution)

1. *If* $decs, var{:}class, decs' \vdash con_1 \equiv con_2 : knd$ *and* $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}con_1 \equiv \{val/var\}con_2 : knd$.

2. *If* $decs, var{:}class, decs' \vdash sig_1 \equiv sig_2 : \mathsf{Sig}$ *and* $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}sig_1 \equiv \{val/var\}sig_2 : \mathsf{Sig}$.

3. *If* $decs, var{:}class, decs' \vdash sig_1 \leq sig_2 : \mathsf{Sig}$ *and* $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}sig_1 \leq \{val/var\}sig_2 : \mathsf{Sig}$.

4. *If* $decs, var{:}class, decs' \vdash exp : con$ *and* $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}exp : \{val/var\}con$.

5. *If* $decs, var{:}class, decs' \vdash mod : sig$ *and* $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}mod : \{val/var\}sig$.

6. *If* $decs, var{:}class, decs' \vdash con : knd$ *and* $decs \vdash val : sig'$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}con : knd.$

7. *If* $decs, var{:}class, decs' \vdash exp \downarrow con$ *and* $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}exp \downarrow \{val/var\}con.$

8. *If* $decs, var{:}class, decs' \vdash mod \downarrow sig$ *and* $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}mod \downarrow \{val/var\}sig.$

Proof (sketch): [By induction on the derivation of the first premises; due to space constraints we show only two sample cases]

- Case: The derivation ends with Typing Rule 49:

  $$\frac{\begin{array}{c} decs, var{:}class, decs' \vdash exp : con_2 \rightharpoonup con \\ decs, var{:}class, decs' \vdash exp' : con_2 \end{array}}{decs, var{:}class, decs' \vdash exp\ exp' : con}$$

  and $decs \vdash val : class$. By the inductive hypothesesΓ

  $$\begin{array}{r} decs, \{val/var\}decs' \vdash \{val/var\}exp : \\ (\{val/var\}con_2) \rightharpoonup (\{val/var\}con) \end{array}$$

  and

  $$decs, \{val/var\}decs' \vdash \{val/var\}exp' : \{val/var\}con_2.$$

  Therefore from Typing Rule 49Γwe can conclude

  $$decs, \{val/var\}decs' \vdash \{exp_{\mathrm{v}}/var\}(exp\ exp') : con.$$

- Case: the derivation ends with Typing Rule 31:

  $$\frac{\begin{array}{c} decs, var{:}class, decs' \vdash mod_{\mathrm{v}} : [lab_1{:}dec_1, \cdots, lab_m{:}dec_m, \\ \hfill lab{:}var{:}knd{=}con_2, sdecs] \\ decs, var{:}class, decs', dec_1, \cdots, dec_m \vdash con \equiv con_2 : knd \\ decs, var{:}class, decs' \vdash con : knd \end{array}}{decs, var{:}class, decs' \vdash mod_{\mathrm{v}}.lab \equiv con : knd}$$

  and $decs \vdash val : class$. By the using properties 5Γ1Γand 6 of the induction hypothesisΓand Typing Rule 31Γwe have

  $$decs, \{val/var\}decs' \vdash \{val/var\}mod_{\mathrm{v}}.lab \equiv \{val/var\}con : knd.$$

■

Claim 3 (Decidability)
All internal language judgments considered by the elaborator are decidable.

Proof (sketch): Because all internal-language module expressions generated by the elaborator have most-specific signaturesΓthe decidability of the static semantics can be reduced in a straightforward fashion to the decidability of constructor equivalence. Of the constructor equivalence judgmentsΓRules 32–45 describe a familiar simply-typed lambda calculus with records and constants. Even allowing the use of simple abbreviations in the context (Rule 30) can be seen to preserve decidability. Roughly speaking one can always first "inline" the abbreviations and then test for equivalence as in the previous case.

It is not obvious how to extend the proof to include Rule 31 in a simple fashion. Lillibridge [Lil97] gives a complex argument for the decidability of constructor equivalence in a related system (without records of constructors). We conjecture an analogous argument would apply for our system. ■

7.1.2 Dynamic Semantics

We define two states to be *equivalent*Γwritten

$$(\Delta, \sigma, E, \mathit{phrase}) \cong (\Delta', \sigma', E', \mathit{phrase}'),$$

if the two states are equal componentwise up to consistent renaming of the locations and exception tags appearing in Δ and normal renaming of bound-variables. This is clearly an equivalence relation on states which preserves the property of being a terminal state.

Lemma 4 (Determinacy of Evaluation)
The following properties hold:

1. *If Σ is terminal and $\Sigma \cong \Sigma'$, then Σ' is also terminal.*

2. *If $\Sigma \hookrightarrow \Sigma_1$ and $\Sigma \cong \Sigma'$, then there exists a state $\Sigma_1' \cong \Sigma_1$ such that $\Sigma' \hookrightarrow \Sigma_1'$.*

3. *If $\Sigma_1 \hookrightarrow \Sigma_1'$, $\Sigma_2 \hookrightarrow \Sigma_2'$ and $\Sigma_1 \cong \Sigma_2$ then $\Sigma_1' \cong \Sigma_2'$.*

4. *If $\Sigma \hookrightarrow^* \Sigma_t$ and $\Sigma \hookrightarrow^* \Sigma_t'$ then $\Sigma_t \cong \Sigma_t'$.*

A context C is a phrase with a single *hole*Γwritten []. We write $C[\mathit{phrase}]$ for the result of filling the hole in C with *phrase*Γpossibly incurring variable capture. A stack of frames E determines a context $\widehat{E}$ in the obvious way: $\widehat{[]}$ is the context []Γand $\widehat{E \circ F}$ is the context $\widehat{E}[F]$.

Proposition 5 (Decomposition & Replacement)

1. *If $\mathit{decs} \vdash \widehat{E}[\mathit{exp}] : \mathit{con}$ and exp is closed, then $\mathit{decs} \vdash \mathit{exp} : \mathit{con}'$ for some type con'. Furthermore, if $\mathit{decs} \vdash \mathit{exp}' : \mathit{con}'$ where exp' is closed, then $\mathit{decs} \vdash \widehat{E}[\mathit{exp}'] : \mathit{con}$.*

2. *If $\mathit{decs} \vdash \widehat{E}[\mathit{mod}] : \mathit{con}$ and mod is closed, then $\mathit{decs} \vdash \mathit{mod} : \mathit{sig}$ for some signature sig. Furthermore, if $\mathit{decs} \vdash \mathit{mod}' : \mathit{sig}$ where mod' is closed, then $\mathit{decs} \vdash \widehat{E}[\mathit{mod}'] : \mathit{con}$.*

3. *If $\mathit{decs} \vdash \widehat{E}[\mathit{con}'] : \mathit{con}$ and con' is closed, then $\mathit{decs} \vdash \mathit{con} : \mathit{knd}$ for some kind knd. Furthermore, if $\mathit{decs} \vdash \mathit{con}'' : \mathit{knd}$ where con'' is closed, then $\mathit{decs} \vdash \widehat{E}[\mathit{con}''] : \mathit{con}$.*

Proof (sketch): [By induction on E]

If $E = []$ then the proposition is trivialΓsince $\widehat{E}[phrase] = phrase$. HenceΓassume $E = E' \circ F$. Due to space constraintsΓwe show just one typical case.

- Case: $F = []\ exp_2$. Then $\widehat{E}[exp] = \widehat{E'}[exp\ exp_2]$. By the inductive hypothesisΓwe have $decs \vdash exp\ exp_2 : con'$ for some con'. Inspection of the typing rules shows that this derivation must contain an application of Rule 49 or Rule 50; we show here only the former caseΓthe other is similar. By inversion $decs \vdash exp : con_2 \rightharpoonup con'$ and $decs \vdash exp_2 : con_2$ for some type con_2. FurthermoreΓif $decs \vdash exp' : con_2 \rightharpoonup con'$ then $decs \vdash exp'\ exp : con'$ and by the inductive hypothesis we have $decs \vdash \widehat{E'}[exp'\ exp_2] : con$; that isΓ$decs \vdash \widehat{E}[exp'] : con$.

■

7.1.3 Soundness of the Internal Language

Following Harper [Har93]Γwe say that a store σ is well-formed with respect to a context ΔΓ written $\Delta \vdash \sigma$Γif

$$\forall loc \in \mathrm{BV}(\Delta),\ \text{if } \Delta \vdash loc : con\,\mathsf{Ref} \text{ then } \Delta \vdash \sigma(x) : con.$$

This formulation of store typing avoids the need for complex maximal fixed point constructions [Tof90]. (An essentially similar observation was made by Wright and Felleisen [WF91].)

Fix a base type *ans* of *answers* to which a completeΓclosed program might evaluate.[2] We can say that a machine state is well-formedΓwritten

$$\vdash (\Delta, \sigma, E, phrase),$$

if and only if $\Delta \vdash \widehat{E}[phrase] : ans$Γ*phrase* is closedΓand $\Delta \vdash \sigma$. An important property of the internal language semantics is that well-formedness of a state is preserved by evaluation.

Proposition 6 (Preservation)
If $\vdash (\Delta, \sigma, E, phrase)$ *and*

$$(\Delta, \sigma, E, phrase) \hookrightarrow (\Delta', \sigma', E', phrase')$$

then $\vdash (\Delta', \sigma', E', phrase')$.

Proof (sketch):
If the transition is a search rule then preservation follows triviallyΓfor $\Delta = \Delta'$Γ$\sigma = \sigma'$Γand $\widehat{E}[phrase] = \widehat{E'}[phrase']$. Hence we need only consider the reduction rules. We show only one sample case here.

[2] A reasonable choice might be String, or Unit if we model all I/O by updating the store; the particular choice does not affect our results.

- Case: Transition Rule 160:

$$\begin{array}{l}(\Delta, \sigma, E \circ (\pi_{lab}\,[]), \{rbnds_{\mathrm{v}}, lab{=}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\}) \hookrightarrow \\ \quad (\Delta, \sigma, E, exp_{\mathrm{v}})\end{array}$$

  Then $\widehat{E \circ (\pi_{lab}}\,[])[\{rbnds_{\mathrm{v}}, lab{=}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\}] = \widehat{E}[\pi_{lab}\,\{rbnds_{\mathrm{v}}, lab{=}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\}]$. By the well-formedness assumption and DecompositionΓ $decs \vdash \pi_{lab}\,\{rbnds_{\mathrm{v}}, lab{=}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\} : con'$ for some type con'. Since this judgement must be proven using Typing Rules 54 and 53Γby inversion we have that $decs \vdash exp_{\mathrm{v}} : con'$. Since exp_{v} is must by closed by the well-formedness assumption and $decs \vdash \widehat{E}[exp_{\mathrm{v}}] : ans$ by ReplacementΓwell-formedness is preserved.

■

FurthermoreΓevaluation of a well-typed program can never "get stuck": if a well-formed state is not terminalΓthere is always an applicable transition to another (well-formed) state. The proof relies on a characterization of the shapes of closed values of each type.

Lemma 7 (Canonical Forms)

1. *Assume* $\Delta \vdash exp_{\mathrm{v}}' : con'$ *where* exp_{v}' *is closed.*

If con' *is of the form...*	*then* exp_{v}' *is of the form...*
$con_1 \rightharpoonup con_2$	$\pi_{\overline{k}}\,\mathsf{fix}\,fbnds\,\mathsf{end}$
$con_1 \rightarrow con_2$	$\pi_{\overline{1}}\,\mathsf{fix}\,fbnd\,\mathsf{end}$
$\{lab_1{:}con_1, \cdots, lab_n{:}con_n\}$	$\left\{\begin{array}{l}\{lab_1{=}exp_{\mathrm{v}1}, \cdots, lab_n{=}exp_{\mathrm{v}n}\} \\ \mathsf{fix}\,fbnds\,\mathsf{end}\end{array}\right.$
$\Sigma_{\langle lab_i \rangle}\,(lab_1{\mapsto}con_1, \ldots, lab_n{\mapsto}con_n)$	$\mathsf{inj}_{lab_i}^{\Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)}\ exp_{\mathrm{v}}$
$(\pi_i\,(\mu\,con))\ con'$	$\mathsf{roll}^{(\pi_i\,(\mu\,con))\ con'}\ exp_{\mathrm{v}}$
Tagged	$\mathsf{tag}(tag, exp_{\mathrm{v}})$
con Ref	loc
base type	$scon$

2. *Assume* $\Delta \vdash mod_{\mathrm{v}}' : sig'$ *where* mod_{v}' *is closed.*

If sig' *is of the form...*	*then* mod_{v}' *is of the form...*
$[sdecs]$	$[sbnds_{\mathrm{v}}]$
$var{:}sig \rightharpoonup sig'$	$\lambda var{:}sig.mod$
$var{:}sig \rightarrow sig'$	$\lambda var{:}sig.mod$

Proof: By inspection of the applicable typing rules for values. For exampleΓconsider the case in which $\Delta \vdash exp_{\mathrm{v}}' : con'$ where con' is a record type. There are numerous rules that allow the conclusion that an expression has a record type (applicationΓrecord projectionΓ etc.) but only one—the rule for record expressions—applies in the case of a closed value. This is the only possible form for exp_{v}. The other cases are analogous. ■

Proposition 8 (Progress)
If $\vdash \Sigma$ *then either* Σ *is terminal or there exists a state* Σ' *such that* $\Sigma \hookrightarrow \Sigma'$.

Proof (sketch):
There are three cases:

1. Σ is terminal.

2. $\Sigma = (\Delta, \sigma, \mu, E, \mathit{phrase})$ where *phrase* is not a value. By inspectionΓfor every possible phrase syntax there is an applicable rule in the dynamic semantics (a reduction rule for new_tag[*con*] and search rules otherwise).

3. $\Sigma = (\Delta, \sigma, \mu, E \circ F, \mathit{val})$. This has one subcase for each possible frame F. We show one such case:

   - Case: $F = (\mathsf{get}\,[])$. By well-formedness and DecompositionΓ$\Delta \vdash \mathsf{get}\, \mathit{val} : \mathit{con}'$ for some type con'. By inversion of Rule 59 we have that $\Delta \vdash \mathit{val} : \mathit{con}'\,\mathsf{Ref}$. By Canonical FormsΓ*val* is a locationΓand by well-formedness $\mathit{val} \in \mathrm{BV}(\sigma)$. ThereforeΓTransition Rule 163 applies.

■

7.2 Properties of the Elaborator

The minimal requirement for the elaborator is that the elaboration of EL code yields well-formed IL code:

Proposition 9 (Well-formed translation)
Assume $\vdash \Gamma$ *ok. Then the following propositions hold:*

1. *If* $\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}$ *then* $\Gamma \vdash \mathit{exp} : \mathit{con}$.

2. *If* $\Gamma \vdash \mathit{match} \rightsquigarrow \mathit{exp} : \mathit{con}$ *then* $\Gamma \vdash \mathit{exp} : \mathit{con}$.

3. *If* $\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

4. *If* $\Gamma \vdash \mathit{strexp} \rightsquigarrow \mathit{mod} : \mathit{sig}$ *then* $\Gamma \vdash \mathit{mod} : \mathit{sig}$.

5. *If* $\Gamma \vdash \mathit{strbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

6. *If* $\Gamma \vdash \mathit{funbind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

7. *If* $\Gamma \vdash \mathit{spec} \rightsquigarrow \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sdecs}$ *ok.*

8. *If* $\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}$ *then* $\Gamma \vdash \mathit{sig} : \mathsf{Sig}$.

9. *If* $\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega$ *then* $\Gamma \vdash \mathit{con} : \Omega$.

10. *If* $\Gamma \vdash \mathit{tybind} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

11. *If* $\Gamma \vdash \mathit{typdesc} \leadsto \mathit{sbnds} : \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

12. *If* $\Gamma \vdash \mathit{etypdesc} \leadsto \mathit{sbnds} : \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

13. *If* $\Gamma \vdash \mathit{datbind} \leadsto \mathit{sbnds} : \mathit{sdecs}$ *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

14. *If* $\mathit{decs} \vdash_{\mathrm{inst}} \leadsto \mathit{mod} : \mathit{sig}$ *then* $\mathit{decs} \vdash \mathit{mod} : \mathit{sig}$. *Furthermore, mod is a syntactic value and sig is the most-specific signature of mod.*

15. *If* $\mathit{decs} \vdash_{\mathrm{eq}} \mathit{con} \leadsto \mathit{exp}$ *then* $\mathit{decs} \vdash \mathit{exp} : \mathit{con} \times \mathit{con} \rightharpoonup \mathsf{Bool}$. *Furthermore, exp is a syntactic value. If* $\mathit{decs} \vdash_{\mathrm{eq}} \Lambda(\mathit{var}_1, \ldots, \mathit{var}_n).\mathit{con} \leadsto \mathit{mod}$ *then* $\mathit{decs} \vdash \mathit{mod} : \forall \mathit{var}^{\langle \mathrm{eq} \rangle_1}, \ldots, \mathit{var}^{\langle \mathrm{eq} \rangle_n}.\mathit{con} \times \mathit{con} \rightharpoonup \mathsf{Bool}$.

16. *If* $\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}$ else $\mathit{sbnds} \leadsto \mathit{sdecs}$:, $\Gamma \vdash \mathit{exp} : \mathsf{Tagged}$ *and* $\Gamma \vdash \mathit{exp} : \mathit{con}$, *then* $\Gamma \vdash \mathit{sbnds} : \mathit{sdecs}$.

17. *If* $\mathit{decs}; \mathit{path}{:}\mathit{sig}_0 \vdash_{\mathrm{sub}} \mathit{sdec} \leadsto \mathit{sbnd} : \mathit{sdec}'$ *then* $\mathit{decs} \vdash \mathit{path} : \mathit{sig}_0$, $\mathit{decs} \vdash \mathit{sdec}$ *ok, and* $\mathit{decs} \vdash \mathit{sbnd} : \mathit{sdec}'$.

18. *If* $\mathit{decs} \vdash_{\mathrm{sub}} \mathit{path} : \mathit{sig}_0 \preceq \mathit{sig} \leadsto \mathit{mod} : \mathit{sig}'$ *then* $\mathit{decs} \vdash \mathit{path} : \mathit{sig}_0$, $\mathit{decs} \vdash \mathit{sig}_0 \leq \mathit{sig} : \mathsf{Sig}$, $\mathit{decs} \vdash \mathit{mod} : \mathit{sig}'$, *and* $\mathit{decs} \vdash \mathit{sig}' \leq \mathit{sig} : \mathsf{Sig}$.

19. *If* $\Gamma \vdash_{\mathrm{ctx}} \mathit{labs} \leadsto \mathit{path} : \mathit{class}$ *then* $\Gamma \vdash \mathit{path} : \mathit{class}$. *Furthermore,* $\Gamma \vdash \mathit{path} : \mathit{class}'$ *if and only if* $\Gamma \vdash_{\mathrm{ctx}} \mathit{labs} \leadsto \mathit{path} : \mathit{class}$.

20. *If* $\Gamma; \mathit{path}{:}\mathit{sig} \vdash_{\mathrm{sig}} \mathit{labs} \leadsto \mathit{labs} : \mathit{class}$ *then* $\Gamma \vdash \mathit{path}.\mathit{labs} : \mathit{class}$. *Furthermore, if* $\Gamma \vdash \mathit{path}.\mathit{labs} : \mathit{class}'$ *then* $\Gamma; \mathit{path}{:}\mathit{sig} \vdash_{\mathrm{sig}} \mathit{labs} \leadsto \mathit{labs} : \mathit{class}'$.

21. *If* $\mathit{sig} \vdash_{\mathrm{wt}} \mathit{labs} := \mathit{con} : \mathit{knd} \leadsto \mathit{sig}' : \mathsf{Sig}$, $\Gamma \vdash \mathit{sig} : \mathsf{Sig}$, *and* $\Gamma \vdash \mathit{con} : \mathit{knd}$, *then* $\Gamma \vdash \mathit{sig}' : \mathsf{Sig}$.

22. *If* $\mathit{sig} \vdash_{\mathrm{sh}} \mathit{labs} := \mathit{labs}' : \mathit{knd} \leadsto \mathit{sig}' : \mathsf{Sig}$ *and* $\Gamma \vdash \mathit{sig} : \mathsf{Sig}$ *then* $\Gamma \vdash \mathit{sig}' : \mathsf{Sig}$.

Proof: This follows by induction on the derivation of the first premises. ■

To claim that we have *defined* a languageΓwe need elaboration to be a "function" in the sense that all possible elaborations of an EL program yield "equivalent" IL programs. We argue that such coherence is plausibleΓalso we have not attempted a formal proof.

Claim 10 (Coherence)
If

$$\mathsf{basis}^{*} \rhd \mathit{basis}{:}\mathit{sig}_{\mathit{basis}} \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{ans}$$

and

$$\mathsf{basis}^{*} \rhd \mathit{basis}{:}\mathit{sig}_{\mathit{basis}} \vdash \mathit{expr} \leadsto \mathit{exp}' : \mathit{ans}$$

and

$$(\cdot, \cdot, \mathsf{let}\ \mathit{basis}{=}\mathit{mod}_{\mathit{basis}}\ \mathsf{in}\ \mathit{exp}\ \mathsf{end}) \hookrightarrow^{*} \Sigma_t$$

then for some terminal state $\Sigma_t' \cong \Sigma_t$,

$$(\cdot, \cdot, \mathsf{let}\ \mathit{basis}{=}\mathit{mod}_{\mathit{basis}}\ \mathsf{in}\ \mathit{exp}'\ \mathsf{end}) \hookrightarrow^{*} \Sigma_t'.$$

Proof (sketch): We enumerate the instances of non-determinism in the elaboratorΓand argue that these should not cause incoherence.

1. The elaborator may "guess" types for bound variables (in `fn` expressions) and when instantiating polymorphism. Types in "dead" code may be underconstrainedΓas in

   ```
   (fn x => 3)(fn y => y)
   ```

   where there are infinitely many choices for the type of the identity `fn y => y`. HoweverΓthe dynamic semantics is sufficiently abstract that choices of types do not affect the course of evaluation.

2. SimilarlyΓthe elaborator may have a choice regarding how to resolve overloading. The simplest argument is that each overloaded operator corresponds to a single operator in the dynamic semanticsΓso that again evaluation does not depend on the resolved type. (A more sophisticated argument should be able to show that in a complete programΓ whenever there is a choice of how to resolve overloadingΓthe operator will never be applied.)

3. The elaborator must decide whether valuable expressions should be monomorphic or polymorphicΓand in the latter caseΓon how many type variables to generalize. This turns expressions into functor abstractionsΓwhich in general causes incoherence by changing the order of evaluation—an expression wrapped by a functor will not be evaluated until the functor is applied. HoweverΓwe only generalize valuable expressionsΓ which have no effectsΓso that these choices will not affect observable behavior.

4. There is nondeterminism in the creation of equality functions: there may be different definitions of equality at the same type. HoweverΓwe maintain the invariant that equality functions always perform structural equality on the underlying values regardless of abstraction. Any two candidate definitions then have the same behaviorΓand no incoherence can arise.

■

8 Acknowledgments

We would like to thank Perry Cheng and Martin Elsman for their particularly helpful comments on the work presented here.

References

[Har93] Robert Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169ΓSchool of Computer ScienceΓCarnegie Mellon UniversityΓ 1993.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*Γpages 123–137Γ1994.

[HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*Γ 15(2):211–252Γ 1993.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages*Γ pages 130–141Γ1995.

[HMM90] Robert HarperΓ John C. MitchellΓ and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th Symposium on Principles of Programming Languages*Γpages 341–354Γ1990.

[Ler94] Xavier Leroy. Manifest typesΓmodulesΓand separate compilation. In *21st Symposium on Principles of Programming Languages*Γpages 109–122Γ1994.

[Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*Γ6(5):667–698Γ1996.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesisΓSchool of Computer ScienceΓCarnegie Mellon UniversityΓ 1997. Available as CMU Technical Report CMU-CS-97-122.

[Mor95] Greg Morrisett. *Compiling with Types.* PhD thesisΓSchool of Computer ScienceΓ Carnegie Mellon UniversityΓ1995. Available as CMU Technical Report CMU-CS-95-226.

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*Γpages 409–423. Springer-VerlagΓLNCS 788ΓApril 1994.

[MTH90] Robin MilnerΓMads TofteΓand Robert Harper. *The Definition of Standard ML.* MIT PressΓ1990.

[MTHM97] Robin MilnerΓMads TofteΓRobert HarperΓand Dave MacQueen. *The Definition of Standard ML (Revised).* MIT PressΓ1997.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19ΓAarhus UniversityΓSeptember 1981.

[Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesisΓSchool of Computer ScienceΓ Carnegie Mellon UniversityΓ1996.

[TMC$^+$96] David TarditiΓGreg MorrisettΓPerry ChengΓChris StoneΓRobert HarperΓand Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*Γpages 181–192Γ1996.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*Γ89(1):1–34ΓNovember 1990.

[WF91] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160ΓDepartment of Computer ScienceΓRice UniversityΓ 1991.